



# The GraphQL Guide

It's the new REST ✨



[http://  
graphql.guide](http://graphql.guide)



John Resig



Loren Sands-Ramshaw

---

# Table of Contents

Preface	1.1
Introduction	1.2
Who is this book for?	1.2.1
Background	1.2.2
The book	1.2.3
The code	1.2.4
Git	1.2.4.1
Formatting	1.2.4.2
Resources	1.2.5
Version	1.2.6
Background	1.3
JavaScript	1.3.1
JavaScript classes	1.3.2
JSON	1.3.3
Git	1.3.4
Node, npm, and nvm	1.3.5
HTTP	1.3.6
Server	1.3.7
Databases	1.3.8
MongoDB	1.3.8.1
Redis	1.3.8.2
SQL	1.3.8.3
SPA	1.3.9
SSR	1.3.10
React	1.3.11
Latency	1.3.12
CDN	1.3.13
Webhooks	1.3.14
Testing	1.3.15
Continuous integration	1.3.16
Authentication	1.3.17
Tokens vs. sessions	1.3.17.1
localStorage vs. cookies	1.3.17.2
Browser performance	1.3.18
Chapter 1: Understanding GraphQL Through REST	1.4
Introduction	1.4.1
GraphQL as an alternative to a REST API	1.4.2



---

A simple REST API server	1.4.3
A simple GraphQL server	1.4.4
Querying a set of data	1.4.5
Filtering the data	1.4.6
Async data loading	1.4.7
Multiple types of data	1.4.8
Security & error handling	1.4.9
Tying this all together	1.4.10

## Part I: The Spec

Chapter 2: Query Language	2.1
Document	2.1.1
Fields	2.1.2
Arguments	2.1.3
Fragments	2.1.4
Variables	2.1.5
Directives	2.1.6
Mutations	2.1.7
Subscriptions	2.1.8
Chapter 3: Type System	3.1
Schema	3.1.1
Scalar types	3.1.2
Enum types	3.1.3
Object types	3.1.4
Query & Mutation types	3.1.5
Lists	3.1.6
Non-null	3.1.7
Arguments	3.1.8
Unions	3.1.9
Interfaces	3.1.10
Chapter 4: Validation & Execution	4.1
Validation	4.1.1
Resolvers	4.1.2
Execution	4.1.3
Error handling	4.1.4

## Part II: The Client

Chapter 5: Client Dev	5.1
Anywhere: HTTP	5.1.1

---

---

cURL	5.1.1.1
JavaScript	5.1.1.2
Client libraries	5.1.2
Streamlined request function	5.1.2.1
Typing	5.1.2.2
View layer integration	5.1.2.3
Caching	5.1.2.4
DevTools	5.1.2.5
Chapter 6: React	6.1
Setting up	6.1.1
Build options	6.1.1.1
App structure	6.1.1.2
Set up Apollo	6.1.1.3
Querying	6.1.2
First query	6.1.2.1
Loading	6.1.2.2
Polling	6.1.2.3
Subscriptions	6.1.2.4
Lists	6.1.2.5
Query variables	6.1.2.6
Skipping queries	6.1.2.7
Authentication	6.1.3
Logging in	6.1.3.1
Resetting	6.1.3.2
Mutating	6.1.4
First mutation	6.1.4.1
Listing reviews	6.1.4.2
Optimistic updates	6.1.4.3
Arbitrary updates	6.1.4.4
Creating reviews	6.1.4.5
Using fragments	6.1.4.6
Deleting	6.1.4.7
Error handling	6.1.4.8
Editing reviews	6.1.4.9
Advanced querying	6.1.5
Paginating	6.1.5.1
Offset-based	6.1.5.1.1
page	6.1.5.1.1.1
skip & limit	6.1.5.1.1.2
Cursors	6.1.5.1.2

---

after	6.1.5.1.2.1
orderBy	6.1.5.1.2.2
Updating multiple queries	6.1.5.2
Local state	6.1.5.3
Direct writes	6.1.5.3.1
Local mutations	6.1.5.3.2
REST	6.1.5.4
Review subscriptions	6.1.5.5
Subscription component	6.1.5.5.1
Add new reviews	6.1.5.5.2
Update on edit and delete	6.1.5.5.3
Prefetching	6.1.5.6
On mouseover	6.1.5.6.1
Cache redirects	6.1.5.6.2
Batching	6.1.5.7
Persisting	6.1.5.8
Multiple endpoints	6.1.5.9
Extended topics	6.1.6
Linting	6.1.6.1
Setting up linting	6.1.6.1.1
Fixing linting errors	6.1.6.1.2
Using linting	6.1.6.1.3
Uploading files	6.1.6.2
Testing	6.1.6.3
Chapter 7: Vue	7.1
Chapter 8: React Native	8.1
Chapter 9: iOS	9.1
Chapter 10: Android	10.1

## Part III: The Server

Chapter 11: Server Dev	11.1
Introduction	11.1.1
Why build a GraphQL server?	11.1.1.1
What kind of GraphQL server should I build?	11.1.1.2
Building	11.1.2
Project setup	11.1.2.1
Types and resolvers	11.1.2.2
Authentication	11.1.2.3
Data sources	11.1.2.4

Setting up	11.1.2.4.1
File structure	11.1.2.4.2
Creating reviews	11.1.2.4.3
Custom scalars	11.1.2.5
Creating users	11.1.2.6
Protecting with secret key	11.1.2.6.1
Setting user context	11.1.2.6.2
Linking users to reviews	11.1.2.6.3
Authorization	11.1.2.7
Errors	11.1.2.8
Nullability	11.1.2.8.1
Union errors	11.1.2.8.2
formatError	11.1.2.8.3
Logging errors	11.1.2.8.3.1
Masking errors	11.1.2.8.3.2
Error checking	11.1.2.8.4
Custom errors	11.1.2.8.5
Subscriptions	11.1.2.9
githubStars	11.1.2.9.1
reviewCreated	11.1.2.9.2
Testing	11.1.3
Static testing	11.1.3.1
Review integration tests	11.1.3.2
Code coverage	11.1.3.3
User integration tests	11.1.3.4
Unit tests	11.1.3.5
End-to-end tests	11.1.3.6
Production	11.1.4
Deployment	11.1.4.1
Options	11.1.4.1.1
Deploying	11.1.4.1.2
Environment variables	11.1.4.1.3
Database hosting	11.1.4.2
MongoDB hosting	11.1.4.2.1
Redis hosting	11.1.4.2.2
Redis PubSub	11.1.4.2.2.1
Redis caching	11.1.4.2.2.2
Querying in production	11.1.4.3
Analytics	11.1.4.4
Error reporting	11.1.4.5

---

More data sources	11.1.5
SQL	11.1.5.1
SQL setup	11.1.5.1.1
SQL data source	11.1.5.1.2
SQL testing	11.1.5.1.3
SQL performance	11.1.5.1.4
REST	11.1.5.2
GraphQL	11.1.5.3
Custom data source	11.1.5.4
Prisma	11.1.5.5
Extended topics	11.1.6
Mocking	11.1.6.1
Pagination	11.1.6.2
Offset-based	11.1.6.2.1
Cursors	11.1.6.2.2
after an ID	11.1.6.2.2.1
Relay cursor connections	11.1.6.2.2.2
File uploads	11.1.6.3
Client-side	11.1.6.3.1
Server-side	11.1.6.3.2
Schema validation	11.1.6.4
Apollo federation	11.1.6.5
Federated service	11.1.6.5.1
Federated gateway	11.1.6.5.2
Extending entities	11.1.6.5.3
Managed federation	11.1.6.5.4
Deploying federation	11.1.6.5.5
Hasura	11.1.6.6
Schema design	11.1.6.7
One schema	11.1.6.7.1
User-centric	11.1.6.7.2
Easy to understand	11.1.6.7.3
Easy to use	11.1.6.7.4
Mutations	11.1.6.7.5
Arguments	11.1.6.7.5.1
Payloads	11.1.6.7.5.2
Versioning	11.1.6.7.6
Custom schema directives	11.1.6.8
@tshirt	11.1.6.8.1
@upper	11.1.6.8.2

---

---

@auth	11.1.6.8.3
Subscriptions in depth	11.1.6.9
Server architecture	11.1.6.9.1
Subscription design	11.1.6.9.2
Security	11.1.6.10
Auth options	11.1.6.10.1
Authentication	11.1.6.10.1.1
Authorization	11.1.6.10.1.2
Denial of service	11.1.6.10.2
Performance	11.1.6.11
Data fetching	11.1.6.11.1
Caching	11.1.6.11.2
Future	11.1.6.12

---

## Preface

👋 Hello there, dear reader 😊. Welcome to the Guide! We are John and Loren, your authors. John created jQuery, and Loren is slightly less famous but writes good 🍌. We're here to tell you about GraphQL, the system we believe will eclipse REST as the best way to fetch data from servers. We'll get into why in Chapter 1, but for now, here's the story of this book:

GraphQL was released in mid 2015, and its adoption has been accelerating ever since. In 2016, Github [switched its API](#) from REST to GraphQL. In 2017, AWS launched a GraphQL-based [platform](#) for building apps. Both [npm](#) and the [State of JavaScript survey](#) named 2018 the year of GraphQL. John and Loren both realized early on that A) GraphQL was going to be very important for app development, and B) a great book on GraphQL didn't yet exist! We each decided independently to write one, but then we found each other and—over schnitzel at a biergarten in Brooklyn—decided to join forces.

We're writing the complete reference: what GraphQL is, why to use it, and most importantly, how to use it—on the server, in the browser, on mobile, with React, React Native, Vue, Node, Java, and Swift. We'll take you step-by-step through building an app, so you can see the practical need behind each part of GraphQL. We're doing it as an ebook so we can always keep it up to date. A physical book would quickly fall behind best practices in such a fast-moving space.

We'd like to thank everyone who helped out with this book, including our technical reviewers Tom Coleman, Brad Crispin, Abhi Aiyer, Oleksandr Bordun, Heather Armstrong, Justin Krup, Melek Hakim, Kamal Radharamanan, Lewi Gilamichael, and Enno Thoma, our designer Genki Hagata, and our copy editors Lauren Itzla, Rachel Lake, and Paul Ramshaw. We'd also like to thank those who wrote the libraries on which this book is based, especially the GraphQL and Apollo communities.

If you'd like to improve this resource for those who read after you, we welcome your suggestions in the form of GitHub issues or PRs on the [book text repo](#) and the [code repo](#) 🙏.

We've found that building apps with GraphQL is less difficult and more fun, and we think you'll be similarly impressed. We hope you enjoy 😊.

John Resig and Loren Sands-Ramshaw

Brooklyn, New York

June, 2018

---

© 2020 *The GraphQL Guide*

# Introduction

- [Who is this book for?](#)
  - [Background](#)
  - [The book](#)
  - [The code](#)
    - [Git](#)
    - [Formatting](#)
  - [Resources](#)
  - [Version](#)
- 

## Who is this book for?

This book is for most programmers. If you write software that fetches data from a server, or you write server code that provides data to others, this book is for you. It's particularly relevant to frontend and backend web and mobile developers. If you don't know modern JavaScript, we recommend [learning that first](#), but it's not necessary. For example, if you only know Ruby, you can likely follow the JavaScript server code in Chapter 11 well enough to learn the important concepts of a GraphQL server, most of which will apply to using the `graphql` gem in your Ruby server code.

This book will be especially poignant to these groups of people:

- Backend devs who work on REST APIs and who:
  - write a lot of similar code to fetch data and format it into JSON,
  - maintain view- or device-specific endpoints, or
  - have multiple APIs that use overlapping business data.
- Frontend devs who either:
  - don't use a caching library, and manually keep track of what data has already been fetched from the server, or
  - use a cache, and write a lot of code to fetch data over REST and put it in the cache (we're looking at you, Reduxers 🙄😅).

## Background

We have a [Background](#) chapter that provides concise introductions to various background topics. You're welcome to either look through them now or individually as you go along—at the beginning of a section, you'll find a list of topics it assumes knowledge of, like the [Anywhere: HTTP](#) section, which has two listed:

Background: [HTTP](#), [JSON](#)

Some topics like [Git](#) and [Node](#) are necessary for following along with the coding. Others, like [Tokens vs. sessions](#), are nice to know, but not necessary.

## The book

---



While this book is great when read cover-to-cover, it's organized as a reference text, so you can also use it to look up a specific topic. For instance, if you're familiar with most GraphQL types but want to learn about Unions, you can look them up in the Table of Contents under Chapter 3: Type System—[Section 9: Unions](#). Or if you're already doing basic queries in your React app, and you want to implement infinite scrolling, you can look it up under Chapter 6: React—[Section 5: Advanced querying—Paginating](#).

[Chapter 1](#) introduces GraphQL and shows why it's better than REST.

Chapters [2](#) and [3](#) explain the language itself and its type system.

[Chapter 4](#) goes in depth on how a GraphQL server responds to a query. It's great for a full understanding of the technology, but you don't *need* to know it unless you're contributing to a GraphQL server library. So it's totally fine to skip this—you'll still understand everything if you go straight to [Chapter 11](#), the practical server-coding chapter.

[Chapter 5: Client Dev](#) is the first of the coding chapters, and introduces common concepts among client libraries. Then we have a chapter for each library:

- [Chapter 6: React](#)
- [Chapter 7: Vue](#)
- [Chapter 8: React Native](#)
- [Chapter 9: iOS](#)
- [Chapter 10: Android](#)

[Chapter 11: Server Dev](#) is our looooong server-coding chapter 🐞. All of the examples are in Node with the `apollo-server-express` library, but almost all of the concepts can be applied to [other languages' GraphQL libraries](#).

## The code

We intersperse blocks of code throughout the text. When we add code to a file that we've shown previously, we often just display the additions and some context, with ellipses ( `...` ) in place of existing code. These additions will be clearest if you read the book with the code open in another window. Further, we believe humans usually learn better if they write things out themselves, so we encourage you to write out the code for each step, and get it working on your computer before moving on to the next step.

We recommend using Chrome and [VS Code](#).

Code snippets are better formatted and sized in the PDF version of the book. If you're reading this in EPUB or MOBI format on your phone, turning sideways into landscape mode will help reduce code snippet wrapping.

## Git

In Chapters 6–11, you'll learn through writing an app, step by step. Each chapter has its own repository. Each step has a branch in that repo, for example, branch `0` is the starter template, branch `1` has the code you write in step 1, etc. The branches we link to in the text also have a version number, and have the format: `[step]_[version]`. When the first version of the Guide was published, the Chapter 6 code version was `0.1.0`, so step 1 linked to branch `1_0.1.0`. The current version is `0.2.0`, so step 1 links to `1_0.2.0`.

If you skip the beginning of Chapter 6 and go straight to the [Listing reviews](#) section, it says to start with step 9 (`9_0.2.0`). So we can look at the app in that state with these terminal commands:

```
git clone https://github.com/GraphQLGuide/guide.git
cd guide/
git checkout 9_0.2.0
npm install
npm start
```

Check out the [git](#) and [npm](#) background sections if you're unfamiliar with these commands.

If we get stuck, we can look at the diff between step 9 and step 10 with GitHub's compare feature:

```
github.com/[repository]/compare/[tag 1]...[tag 2]
```

which in our case would be:

```
github.com/GraphQLGuide/guide/compare/9_0.2.0...10_0.2.0
```

We can also see the solution to the current step by checking out the next step:

```
git checkout 10_0.2.0
npm start
```

## Formatting

All the JavaScript code is [ES2016](#) and formatted with [prettier](#) with two [settings](#):

```
.prettierrc
```

```
singleQuote: true
semi: false
```

This means `'` instead of `"` for string literals and no unnecessary semicolons.

## Resources

If you run into issues, we recommend posting to Stack Overflow with the relevant tag, for instance [react-apollo](#) for Chapter 6. If you have the Full edition, you can also ask the community in the [#support](#) Slack channel or email the technical support address we gave you.

If the issue is with our code, please search the repository's issues to see if it's an existing bug, and if it's new, submit it! 🙏🙌

[github.com/GraphQLGuide/guide/issues](https://github.com/GraphQLGuide/guide/issues)

Another important resource is the docs! Here they are for each library:

- [Chapter 6 and 8](#): [react-apollo](#)
- [Chapter 7](#): [vue-apollo](#)
- [Chapter 9](#): [Apollo iOS](#)
- [Chapter 10](#): [Apollo-Android](#)
- [Chapter 11](#): [apollo-server-express](#)

## Version

Book version: [r5](#) ([changelog](#))

Published May 23, 2020

As we write more of the book, we'll send you new versions of it (using the email address on the GitHub account you connected when you purchased the book from [graphql.guide](#)).

## Chapter 6

Code version: 0.2.0 ([changelog](#))

```
react-apollo 2.5  
graphql 0.14  
react 16.8
```

## Chapter 11

Code version: 0.2.0 ([changelog](#))

```
apollo-server 2.12.0
```

# Chapter: Background

Chapter contents:

- [JavaScript](#)
- [JavaScript classes](#)
- [JSON](#)
- [Git](#)
- [Node, npm, and nvm](#)
- [HTTP](#)
- [Server](#)
- [Databases](#)
  - [MongoDB](#)
  - [Redis](#)
  - [SQL](#)
- [SPA](#)
- [SSR](#)
- [React](#)
- [Latency](#)
- [CDN](#)
- [Webhooks](#)
- [Testing](#)
- [Continuous integration](#)
- [Authentication](#)
  - [Tokens vs. sessions](#)
  - [localStorage vs. cookies](#)
- [Browser performance](#)

---

This chapter provides concise introductions to various background topics. You're welcome to either read them all up front or individually as you go along—at the beginning of a section, you'll find a list of topics it assumes knowledge of, like the [Anywhere: HTTP](#) section, which has two listed:

Background: [HTTP](#), [JSON](#)

Some topics, like [Git](#) and [Node](#), are necessary for following along with the coding. Others, like [Tokens vs. sessions](#), are nice to know, but not necessary.

## JavaScript

Most of the code in the book is in modern JavaScript. If you're new to JS, you can learn through interactive [courses](#), video ([intro](#) and [intermediate](#)), or [a combination](#).

If you know traditional JS, but some of the new syntax is unfamiliar (for instance [async/await](#)), here's a [course on ES6](#).

## JavaScript classes

---

A **class** is a template for an object. With this class:

```
class Animal {  
  constructor(name) {  
    this.name = name  
  }  
  
  speak() {  
    console.log(`${this.name} makes a noise.`)  
  }  
}
```

We can make an object, or **instance** of the class:

```
const loren = new Animal('Loren')
```

`loren` is now an instance of `Animal`. When JavaScript evaluated `new Animal('Loren')`, it created a new object and called the `constructor` method with the string `'Loren'`, which set the object's property `name` to `'Loren'` and (implicitly) returned the new object. Now when we do:

```
console.log(loren.name)  
loren.speak()
```

We see the output:

```
Loren  
Loren makes a noise.
```

The class `Animal` is a template that we can create multiple different instances of:

```
const loren = new Animal('Loren')  
const graphy = new Animal('Graphy')  
  
loren.speak()  
graphy.speak()
```

Results in:

```
Loren makes a noise.  
Graphy makes a noise.
```

Both of the instances have the `.speak()` method, but they have different values for the `.name` property, so `.speak()` logs different strings.

We can also create **subclasses** by using the syntax `class SubClass extends SuperClass`:

```
class Animal {  
  constructor(name) {  
    this.name = name  
    console.log(`${this.name} is a ${this.constructor.name}.`)  
  }  
  
  speak() {  
    console.log(`${this.name} makes a noise.`)  
  }  
}  
  
class Dog extends Animal {  
  constructor(name) {  
    super(name)  
  }  
}
```

```
        console.log('Subspecies: Canis lupus familiaris.')
    }
}
```

`Dog` is a subclass of `Animal`. `this.constructor.name` is the name of the class ( `'Dog'` if `new Dog()` or `'Animal'` if `new Animal()` ). In its constructor, it calls the superclass's constructor ( `super(name)` ) and then logs. So now if we do:

```
const graphy = new Dog()
console.log(graphy.name)
graphy.speak()
```

We see:

```
Graphy is a Dog.
Subspecies: Canis lupus familiaris.
Graphy
Graphy makes a noise.
```

A subclass can override a superclass's method or define new methods:

```
class Dog extends Animal {
  constructor(name) {
    super(name)
  }

  speak() {
    console.log(`${this.name} barks.`)
  }
}

const loren = new Animal('Loren')
loren.speak()

const graphy = new Dog('Graphy')
graphy.speak()
graphy.sit()
```

```
Loren is a Animal.
Loren makes a noise.
Graphy is a Dog.
Subspecies: Canis lupus familiaris.
Graphy barks.
Graphy sits.
```

If we tried to do `loren.sit()`, we would get an error because `Animal` doesn't have a `.sit()` method:

```
loren.sit()
      ^
TypeError: loren.sit is not a function
```

We can have multiple subclasses, for instance `Rabbit` and `Cat`, and subclasses can have subclasses, for instance `class Lynx extends Cat`.

## JSON

JSON is a file format for data objects. The objects are structured in attribute–value pairs, where the attribute is a string and the value can be one of the following types:

- Number: `1.14`
- String: `"foo"`
- Boolean: `true`
- null: `null` 😊
- Array of other types: `["foo", true, 1.14]`
- Object: `{ "name": "john" }`

In JSON documents, whitespace doesn't matter, and commas go between attribute–value pairs and between items in arrays. Here's an example, formatted with nice whitespace:

```
{
  "authors": [
    {
      "name": "john",
      "wearsGlasses": true
    },
    {
      "name": "loren",
      "wearsGlasses": true
    }
  ]
}
```

It's also valid JSON to have an array at the top level of the document, e.g.:

```
[{ "name": "john" }, { "name": "loren" }]
```

In JavaScript, if we have this document in a string, we can parse it to create a JavaScript object with the same data:

```
const jsObject = JSON.parse(jsonString)
```

When working with raw [HTTP](#) responses that contain a JSON body, we have to use `JSON.parse()` to get the data into an object. But we'll mostly be working with libraries that take care of this step for us.

## Git

[Git](#) is a version control system for saving your code and keeping a history of the changes. Unfamiliar? Try this [interactive tutorial](#)

## Node, npm, and nvm

[Node](#) is what runs JavaScript on a server. [npm](#) is a JavaScript package manager and registry. Their `npm` command-line tool manages the packages (libraries of JavaScript code) that our app depends on, helping us install and upgrade them. Their registry stores the content of the packages and makes them available for download—it has more packages than any other registry in the history of software! We use npm packages both with code that runs on the server in Node and with code that runs on the client—in the browser or in React Native.

We recommend installing Node with `nvm` (the *Node Version Manager*):

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh | bash
$ nvm install node
$ nvm alias default node
```

This installs the latest version of Node. Then in a new terminal window, we can see the version number with:

```
$ node -v
```

We can keep track of which projects use which versions of node by adding a `.nvmrc` file to the root of each project. It contains a version number (like `8` or `8.11.3`) or `node` to use the latest stable version. Then when we switch projects, we run `nvm use` to switch to that project's version of node:

```
$ nvm use
Found '/guide/.nvmrc' with version <8>
Now using node v8.11.3 (npm v5.6.0)
```

`npm` is a command-line tool that is installed along with Node. When we want to use npm packages in our project, we create a `package.json` file in the project's root directory:

```
{
  "name": "my-project",
  "private": true
}
```

Then we install the package with:

```
$ npm install graphql
```

If we're using a recent version of npm (5.0 or higher), the package name and version will now be saved in our `package.json`:

```
{
  "name": "my-project",
  "private": true,
  "dependencies": {
    "graphql": "^0.13.1"
  }
}
```

We see the current package's version, which was `0.13.1` at time of writing. npm packages follow **SemVer**, a convention for version numbering:

```
[major version].[minor version].[patch version]
```

Major version changes mean that the library's API has been changed in an incompatible way—if we write our code to use version `1.0.0` of a library (for example, using the library's function `doThis()`), our code will probably break if we switch to version `2.0.0`. (For example, if the library renamed `doThis` to `doThat`, and our code were still called `doThis()`, we'd get an error.) Minor and patch version changes do not break the API—if we write our code using version `1.0.0` of a library, we can safely upgrade to version `1.0.8` or `1.4.0`.

Minor version changes mean that functionality has been added—if we write our code using version `1.4.0`, it may break if we switch to version `1.3.0`, because it may use a feature introduced in minor version 4. Patch version changes mean that bugs have been fixed—if we switch from `1.0.8` to `1.0.7`, our code may stop working because of the bug that was fixed in patch version 8.

The one exception to the above is that version numbers with a major version of 0 don't have a stable API, so going from `0.0.1` to `0.0.2` could be breaking—as could going from `0.1.0` to `0.2.0`.

A caret `^` before a version number means that our code depends on any version compatible with that number—for example, if we had a dependency `"foo": "^1.4.0"`, our code should work with any versions between `1.4.0` and `2.0.0`, such as `1.4.1` or `1.11.2`.

We can also see that we have a new `node_modules/` folder, and it contains folders with the package code:



```
$ ls node_modules/  
graphql iterall
```

`iterall` was downloaded as well because it is a dependency of `graphql`, which we can see if we look at its `package.json`:

```
$ cat node_modules/graphql/package.json  
{  
  ...  
  "dependencies": {  
    "iterall": "^1.2.0"  
  },  
  "homepage": "https://github.com/graphql/graphql-js",  
  "name": "graphql",  
  "version": "0.13.1"  
}
```

We don't want to save downloaded packages in git, so we exclude it:

```
$ echo 'node_modules/' >> .gitignore
```

If we're not in an existing git repository, we run `git init` to initialize. Then we can save our files with `git add <filename>` and a commit:

```
$ git add package.json .gitignore  
$ git commit -m 'Added the graphql package'
```

When our code is cloned (by others, or by us in the future), there will be no `node_modules/`. If our code is at <https://github.com/me/app>, then we would do:

```
$ git clone https://github.com/me/app.git  
$ cd app  
$ ls -a  
.  .. .git .gitignore package.json
```

We run `npm install` to download all the packages listed in `package.json` into `node_modules/`:

```
$ npm install  
added 2 packages in 1.026s  
$ ls node_modules/  
graphql iterall
```

And then we could use the package in our JavaScript like this:

```
import { graphql } from 'graphql'  
  
...  
  
graphql(schema, query).then(result => {  
  console.log(result);  
})
```

## HTTP

HTTP is a format for sending messages over the internet. It is used on top of two other message formats—IP (which has an *IP address* and routes the message to the right machine) and TCP (which has a port number and resends any messages that are lost in transit). An HTTP message adds a *method* (like `GET` or `POST`), a path (like `/graphql`), headers (like the `Bearer` header we use for [authentication](#)), and a body (where GraphQL queries and responses go).

When we enter a URL like `http://graphql.guide/` into our browser, it goes through these steps:

- Browser asks DNS server what the IP address of `graphql.guide` is.
- DNS server responds with `104.27.191.39`.

We can see for ourselves what the DNS server says using the `nslookup` command:

```
$ nslookup graphql.guide
Server:      8.8.4.4
Address:     8.8.4.4#53

Non-authoritative answer:
Name:   graphql.guide
Address: 104.27.191.39
```

- Browser sends out a message to the internet that looks like this:

```
IP to 104.27.191.39
TCP to port 80
HTTP GET /
```

- Internet routers look at the IP part, see it is addressed to `104.27.191.39`, and pass it off to a router that is closer to `104.27.191.39`.
- The message arrives at `104.27.191.39` (the IP address of the Guide server), which opens the message, sees that it's trying to connect to port 80, and passes the message to whatever server program (in this case, a Node.js process) is listening at the port.

An *IP address* is the number ID of a computer on the internet, and we can think of a *port* as the number of a program running on that computer.

- The server process sees that the client wants to GET `/`, the root path, and sends back an `index.html` to the client.

This sequence is a little simplified—it actually takes a separate round-trip message to set up the TCP connection, and for `graphql.guide`, the client is actually redirected to HTTPS at the beginning, which uses port 443 and sets up an SSL connection before sending HTTP GET `/`.

## Server

The term *server* may refer to:

1. a computer connected to a network (usually the internet)
2. a process running on that computer that listens to one or more ports
3. the group of computers/processes that share the responsibility of handling requests

In web development, servers are usually either static file servers (which serve files like our HTML, images, and JS bundle), application (app) servers (the ones that power our API and that the client talks to) or database servers. *Server-side* either means app servers or everything that's not the client-side (including file, app, and database servers, as well as any other servers they talk to).

# Databases

Databases are organized collections of data stored on a computer. That computer is called a *database server*, and the computer querying the database (usually an app server) is called the *database client*. Different databases organize their data differently, store it differently, and communicate differently. There are two types of database storage: *in-memory* (the data is stored in RAM, and would be lost in the event of a power outage) and *persistent* (the data is stored on disk—a hard drive or SSD). [Redis](#) is primarily used as an in-memory database, whereas [MongoDB](#) and [SQL](#) databases are usually used as persistent databases.

There are two main categories of databases:

- **Relational databases:** These usually use SQL (Structured Query Language), and follow the relational model, with *tables* of *columns*, *rows*, and unique keys. The most popular relational databases are SQLite for development and PostgreSQL for production.
- **Non-relational (NoSQL) databases:** These usually use their own query language, although some (like the [Dgraph](#) graph database and distributed [FaunaDB](#)) support GraphQL as a way to query the database! 😊 There are a few categories of NoSQL databases:
  - **Document databases** like MongoDB
  - **Graph databases** like [Neo4J](#)
  - **Key-value databases** like Redis
  - **Wide-column databases** like [Cassandra](#)
  - **Multi-model** which support [multiple data models](#)

In this section we'll look at three databases:

- [MongoDB](#)
- [Redis](#)
- [SQL](#)

## MongoDB

While MongoDB can be used as an in-memory database, it's usually used as a persistent database. The data is organized in *collections* of JSON-like *documents*. Developers communicate with the database using [MongoDB schema statements](#):

```
import { MongoClient } from 'mongodb'

const DATABASE_SERVER_URL = 'mongodb://my-database-server-domain.com:27017/guide'

const client = new MongoClient(DATABASE_SERVER_URL)

const example = async () => {
  await client.connect()
  const db = client.db()

  // get the collection with the name 'users'
  const users = db.collection('users')

  // insert a new user document into the users collection
  await users.insertOne({
    firstName: 'Loren',
    email: 'loren@graphql.guide'
  })

  // update the document where `firstName` is Loren by
  // setting the `lastName` field (a new field)
  await users.updateOne(
    { firstName: 'Loren' },
    {
```

```

    $set: { lastName: 'Sands-Ramshaw' }
  }
)

// fetch the document where `firstName` is Loren
const loren = await users.findOne({ firstName: 'Loren' })
console.log(loren)

users.deleteOne({ _id: loren._id })
}

example()

```

In practice, we should handle errors—either with a try-catch or `.catch ( await users.findOne().catch(e => console.log(e)))` ).

This would log something like:

```

{
  _id: ObjectId('5d24f846d2f8635086e55ed3'),
  firstName: 'Loren',
  lastName: 'Sands-Ramshaw',
  email: 'loren@graphql.guide'
}

```

When a new document is inserted into a collection, if no ID is provided (in the field named `_id` ), then a unique [ObjectId](#) is generated. We usually interact with ObjectIds as strings, but they also encode the creation time, which we can get with `loren._id.getTimestamp()` .

The above code uses the [mongodb module](#), which is the official Node.js driver provided by MongoDB. It's always up to date with security patches, it supports the latest MongoDB versions, and it includes support for:

- [Transactions](#)
- Aggregations ([collection](#) and [database](#) level)
- Retryable reads and writes
- [Client-side field-level encryption](#)

Querying with `mongodb` is through MongoDB schema statements. It can be simplified in some ways with the [mongoose module](#), the main JavaScript *ORM* for MongoDB. We'll use Mongoose in [Chapter 1](#) and `mongodb` in [Chapter 11: Server Dev](#).

An *ORM*, or object-relational mapping, is a library that models database records as objects. In the case of Mongoose, it models MongoDB documents as JavaScript objects. It also does schema validation, type casting, query building, and business logic hooks.

## Redis

[Redis](#) is an in-memory key-value database with optional durability:

- *in-memory*: data is read from and written to memory (RAM) and not durable (data is lost on restart or power loss)
- *key-value*: data is stored in values and fetched by keys (unique strings)
- *optional durability*: data can be periodically persisted (written to disk), thus making almost all the data (minus whatever changed in the last couple seconds since the last write to disk) durable (able to be recovered on restart)

Redis is usually used as a cache—for data that we want quick access to but are okay losing. We can install locally with `brew install redis` and start with `brew services start redis` . Then we can query using the [ioredis](#) npm library:

```
import Redis from 'ioredis'
```

```
const redis = new Redis()

await redis.set('name', 'The Guide')
const name = await redis.get('name')
// 'The Guide'

redis.del('name')
```

This uses the three basic commands: SET, GET, and DEL (delete). Here the value is just a string ( 'The Guide' ), but there are other types of data that values can be, including:

- lists (list of strings, ordered by time of insertion)
- sets (unique, unordered strings)
- sorted sets
- hashes (similar to JS objects)

Hash commands include **HMSET** (hash multiple set) and **HGET** (hash get single field):

```
await redis.hset('latest-review', { stars: '5' text: 'A+' })
const reviewStars = parseInt(await redis.hget('latest-review', 'stars'))
// 5

redis.del('latest-review')
```

## SQL

SQL (Structured Query Language) is a language for querying relational databases like SQLite and PostgreSQL. Relational databases have *tables* instead of MongoDB's collections, and *rows* instead of documents. A row is made up of *values* for each *column* in the table. Columns have a name and a type—for instance a `reviews` table with a column named `star` of type `INTEGER`, which could have a value of `5` in the first row:

--

Unlike MongoDB collections, each table has a schema—its name and list of columns. Both the table schema and query statements are written in SQL. Here are the `CREATE TABLE` and `INSERT` statements to create the pictured table and rows, and `SELECT` to view the table's contents:

```
$ brew install sqlite
$ sqlite3
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> CREATE TABLE reviews(
...>   id INTEGER PRIMARY KEY,
...>   text TEXT NOT NULL,
...>   stars INTEGER
...> );
sqlite> INSERT INTO reviews VALUES(1, 'Breathtaking', 5);
sqlite> SELECT * FROM reviews;
1|Breathtaking|5
sqlite> INSERT INTO reviews VALUES(2, 'tldr', 1);
sqlite> INSERT INTO reviews VALUES(3, "Now that's a downtown job!", null);
sqlite> SELECT * FROM reviews;
1|Breathtaking|5
2|tldr|1
3|Now that's a downtown job!|
```

The `id` column is marked as the `PRIMARY KEY` (each table must have a unique key), and `text` column is non-null ( `NOT NULL` ). `SELECT * from reviews` means "fetch all the values from all the rows in the reviews table," and it prints the results to the console. We insert 3 rows of `VALUES` (the values are listed in the order that the columns are

declared in the schema). The last row is allowed to have a `null` value because the `stars` column wasn't declared with `NOT NULL`. And we see in the final `SELECT` statement result that there's nothing in the third column. There are many other statements and variations to statements. A couple more common ones are `UPDATE` and `DELETE`, which alter and remove rows:

```
sqlite> UPDATE reviews SET stars = 4 WHERE id = 3;
sqlite> SELECT * FROM reviews;
1|Breathtaking|5
2|tldr|1
3|Now that's a downtown job!|4
sqlite> DELETE FROM reviews WHERE stars = 4;
sqlite> SELECT * FROM reviews;
1|Breathtaking|5
2|tldr|1
```

Relational databases have relations between tables—for instance in the reviews table we can have an `author_id` column that matches the `id` column in the users table. When a review row has a value of 1 under its `author_id` column, it means the user row with an `id` of 1 authored the review. We can tell SQL about this relation between the tables by adding this to the reviews table:

```
FOREIGN KEY(author_id) REFERENCES users(id)
```

Then we can make a query that fetches data from both tables using `INNER JOIN`:

```
sqlite> CREATE TABLE users(
...>   id INTEGER PRIMARY KEY,
...>   username TEXT NOT NULL
...> );
sqlite> DROP TABLE reviews;
sqlite> CREATE TABLE reviews(
...>   id INTEGER PRIMARY KEY,
...>   text TEXT NOT NULL,
...>   stars INTEGER,
...>   author_id INTEGER NOT NULL,
...>   FOREIGN KEY(author_id) REFERENCES users(id)
...> );
sqlite> INSERT INTO users VALUES(1, 'lorensr');
sqlite> INSERT INTO reviews VALUES(1, 'Breathtaking', 5, 1);
sqlite> INSERT INTO reviews VALUES(2, 'tldr', 1, 1);
sqlite> SELECT reviews.text, reviews.stars, users.username FROM reviews INNER JOIN users ON reviews.author_id =
users.id;
Breathtaking|5|lorensr
tldr|1|lorensr
```

`Breathtaking|5` is from the reviews table while `lorensr` is from the users table.

While we can send SQL statements as strings in our code, we usually use a library for convenience and security (avoiding [SQL injection](#)). In [Chapter 11: SQL](#) we use the [Knex](#) library, which looks like this:

```
this.knex
  .select('*')
  .from('reviews')
```

## SPA

An [SPA](#) (single-page application) is a website that keeps the same page loaded for the duration of the user's session. Instead of a traditional website, in which every link or button that is clicked causes an [HTTP request](#) to be sent to the server and a new HTML page to be loaded, there is a single HTML page, and JavaScript changes the page to show different views. React, Angular, and Vue are all JS libraries for making SPAs (often called *view libraries*).

## SSR

SSR (server-side rendering) is when, instead of sending a small HTML file and a JS bundle that we ask the client to parse and render into HTML, our server sends fully rendered HTML (that it created by running the JS view code on the server). When that rendered HTML is able to be cached, the client browser can display the page faster than a normal SPA (a normal SPA displays a blank or skeleton HTML page, and then JavaScript constructs the view and puts it on the page). We also have code from our view library that, once the browser loads the static HTML, attaches our app's events handlers (like `onClick`, `onSubmit`, etc.) to the page (through a process called *hydration*).

## React

React was released by Facebook in 2013, and it has since steadily increased in popularity, surpassing Angular in GitHub stars in 2016 to become the most popular JavaScript view library. (And while Vue passed React in star count in 2018, React has 5x the number of npm downloads.) React continues to be developed by a team at Facebook, who have merged in contributions from over one thousand developers.

As a view library, it is responsible for what the user sees on the screen. So its job is putting DOM nodes on the page and updating them. Different view libraries accomplish this in different ways and provide different APIs for us—the developers—to use. The primary features of React are:

- **JSX:** JSX (JavaScript XML) is an extension to JavaScript that allows us to write HTML-like code, with JavaScript expressions inside curly brackets `{}`.
- **Components:** Components are functions or classes that receive arguments (called *props*) and return JSX to be rendered. They can also be used as HTML tags inside JSX: `<div><MyComponent /></div>`.
- **Declarative:** Components automatically get re-run whenever their props or state changes, and the new JSX they return automatically updates the page. This process is called declarative because we declare what our props and state are as well as what the JSX should look like based on those props and state. This is in contrast to an *imperative* view library like jQuery, in which we would make changes to the page ourselves (for example adding an `<li>` to a `<ul>` with `$('#ul').append('<li>New list item</li>')`).
- **Virtual DOM:** React creates a model of the page, and when we return different JSX from our components, React compares the new JSX to the previous JSX, and uses the difference to make the smallest possible changes to the DOM. This process improves the rendering speed.

## Latency

Latency is the delay between when one machine sends a message over the internet and when the other machine receives it. It's usually talked about in terms of round-trip time: the time it takes for the message to get to the destination and for a reply to reach the source. The `ping` command-line tool displays round-trip time between our computer and another machine. Here we see that it takes around 5 milliseconds total for a message to reach the nearest Google server and for the reply to arrive back:

```
$ ping google.com
PING google.com (172.217.10.142): 56 data bytes
64 bytes from 172.217.10.142: icmp_seq=0 ttl=56 time=3.919 ms
64 bytes from 172.217.10.142: icmp_seq=1 ttl=56 time=5.375 ms
64 bytes from 172.217.10.142: icmp_seq=2 ttl=56 time=4.930 ms
64 bytes from 172.217.10.142: icmp_seq=3 ttl=56 time=5.206 ms
64 bytes from 172.217.10.142: icmp_seq=4 ttl=56 time=5.132 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 3.919/4.912/5.375/0.517 ms
```

It generally takes longer to reach servers that are physically farther away. The internet backbone is made of fiber optic cables, and the light messages travelling through them has a maximum speed. It takes 75 ms for a message to go from New York across the Atlantic Ocean to Paris and back. And the same to cross the U.S. to San Francisco and back. 164 ms from New York to Tokyo, and 252 ms from New York to Shanghai.

These numbers will change once [Elon](#) builds [Starlink](#)), a network of near-Earth satellites 🤖. The satellites will be so near that the latency to them from the ground is 7 ms, and then the satellites will communicate with each other by light. Light travels faster in straight lines through space than in cables curved over the Earth's surface, so latency to far-off locations will be reduced!

Why do developers need to know about latency? Because we never want to keep our users waiting! If our web server is in New York, our database is in Shanghai, and our user is in San Francisco, and the request requires 3 database requests in series, and our server code takes 20ms, then the user won't receive a response for  $(75 + 252 * 3 + 20) = 851$  ms! (And this is assuming the [TCP](#) connection is already set up, which would require another round trip from the user to the server, not to mention the longer SSL handshake if it's HTTPS.) Almost one second is a long time for our user, whose human brain [notices delays](#) as short as 100ms. This is why we try to locate our database server in the same data center as our web server (for example both in Amazon's [us-east-1](#) ). It's why we use a [CDN](#) to get our files on servers around the world, closer to our users. It's also why we try to reduce the number of sequential requests we need to make between the client and the server, and why it's so important that we can put all of our queries in a single GraphQL request.

## CDN

A CDN, or *Content Delivery Network*, has servers around the world that deliver our content to users. Because their servers are closer to our users than our servers are, they can respond faster than we can, improving [latency](#). Here is the way they typically deliver our content:

- We tell our domain name registrar (where we bought the domain) to set the CDN as our [DNS](#) server.
- We have our server set a `Cache-Control` header on our responses to HTTP requests. The header tells the CDN how long to serve that response to users.

Then, when a user makes a request, this is what happens the first time:

- The client asks DNS server: "Where is `ourdomain.com/foo` ?"
- The DNS server, which is run by our CDN, replies: "It's at `1.2.3.4` ", which is the IP address of a nearby server run by the CDN.
- The client connects to `1.2.3.4` and makes the request, saying: `GET ourdomain.com/foo` .
- The `1.2.3.4` CDN server doesn't know what the `/foo` response should be, so it makes this request to our server: `GET ourapp.herokuapp.com/foo` .
- The `1.2.3.4` CDN server forwards the response from our server to the client.
- If the response from our server had an HTTP header that says `Cache-Control: max-age=60` , then the CDN caches it for 60 seconds.

After the CDN caches it, during the next minute, here is what happens when other users make the same request:

- The client asks DNS server: "Where is `ourdomain.com/foo` ?"
- The DNS server, which is run by our CDN, replies: "It's at `5.6.7.8` ", which is the IP address of a nearby server run by the CDN.
- The client connects to `5.6.7.8` and makes the request, saying: `GET ourdomain.com/foo` .
- The `5.6.7.8` CDN server finds the `/foo` response in its cache, and sends it to the client.

These subsequent requests take much less time to complete than requests to our server because: A) the CDN servers are closer, so it takes less time to reach them over the internet, and B) the CDN servers have the whole response ready to quickly return, whereas our server would spend time constructing the response.



# Webhooks

Webhooks are a system for how one server can notify another server when something happens: some sites, including GitHub, allow us to provide them with a URL, for instance `https://api.graphql.guide/github-hook`, to which they make an [HTTP](#) request when a certain event occurs. If we tell GitHub we want to know about the [watch event](#) on the Guide repo, then they will send a POST to our server (using the given URL) whenever the repo is starred. The POST will contain a JSON body with information about the event, for example:

```
{
  "action": "started",
  "repository": {
    "name": "guide",
    "full_name": "GraphQLGuide/guide",
    "watchers_count": 9,
    ...
  },
  "sender": {
    "login": "lorensr",
    "type": "User",
    "html_url": "https://github.com/lorensr",
    ...
  }
}
```

Then our server parses the JSON to figure out what happened. In this case, the `sender` is the user who performed the action, and we see under the `repository` attribute that the repo now has 9 watchers.

## Testing

- [Mocking](#)
- [Types of tests](#)

## Mocking

First let's go over *mocking*. Let's say we have a file with math functions:

math.js

```
export const add = (a, b) => a + b
export const multiply = (a, b) => a * b
```

And our app, which uses the math functions:

app.js

```
import { add, multiply } from './math'

export const addThenMultiply = (a, b, c) => multiply(add(a, b), c)
```

A test for our app might look like this:

app.test.js

```
import { addThenMultiply } from './app'

test('returns the correct result', () => {
  const result = addThenMultiply(2, 2, 5)
  expect(result).toEqual(20)
})
```

```
})
```

The test calls `addThenMultiply()`, which then calls `add()` and `multiply()`.

But what if we only want to test the logic inside `addThenMultiply()` and *not* the logic inside `add()` and `multiply()`? Then we need to *mock* `add()` and `multiply()`—replace them with mock functions that either don't do anything, or just return a set number. We replace them with mock functions that don't do anything with `jest.mock()`:

```
import { addThenMultiply } from './app'
import * as math from './math'

jest.mock('./math.js')

test('calls add and multiply', () => {
  addThenMultiply(2, 2, 5)
  expect(math.add).toHaveBeenCalled()
  expect(math.multiply).toHaveBeenCalled()
})
```

`add()` and `multiply()` don't return anything—they just track whether they've been called. So that's what we test. If we want to also test whether they're called in the right way, we can control what they return():

```
import { addThenMultiply } from './app'
import * as math from './math'

jest.mock('./math.js', () => ({
  add: jest.fn(() => 4),
  multiply: jest.fn(() => 20)
}))

test('calls with the right parameters and returns the result of multiply', () => {
  const result = addThenMultiply(2, 2, 5)
  expect(math.add).toHaveBeenCalledWith(2, 2)
  expect(math.multiply).toHaveBeenCalledWith(4, 5)
  expect(result).toEqual(20)
})
```

One danger of mocking too much is that we generally don't want to test the implementation of something—just the output. In this case, `addThenMultiply()` could have been implemented differently, for instance:

app.js

```
import { add } from './math'

export const addThenMultiply = (a, b, c) => {
  const multiplicand = add(a, b)
  let total = 0
  for (let i = 0; i < c; i++) {
    total = add(total, multiplicand)
  }
  return total
}
```

Now even though the function is correct, our test would fail. An example of testing the implementation for React components would be looking at state values instead of just looking at the output (what the render function returns).

## Types of tests

There are three main types of automated tests:

- **Unit:** tests a function (or more generally, a small piece of code) and mocks any functions called by that function.

- **Integration:** tests a function and the functions called by that function, mocking as little as possible. Usually functions that involve network requests are mocked.
- **End-to-end (e2e):** tests the whole running application. Usually refers to the whole stack—including frontend server, API server, and database—running as they would in production, and the tests click and type in the UI. In the context of backend, it can mean just the API server and database are running, and tests send HTTP requests to the API server.

Should we write tests? What kind, and how many?

“Write tests. Not too many. Mostly integration.” —[Guillermo Rauch](#)

Yes, we should write tests. No, we don't need them to cover 100% of our code. Most of our tests should be integration tests.

We write tests so that we can have confidence that when write code, we're not breaking things that used to work. We can cover most of our code (or more importantly, our use cases) with integration tests. Why not cover everything with unit tests? Because it would take forever to write all of them, and some of them would test implementation, so whenever we refactored, we would have to rewrite our tests. We can cover the same amount of code with fewer integration tests, because each test mocks fewer things and covers more code. We don't cover everything with e2e tests because they would take forever to run—after clicking or submitting a form, the test runner has to wait for the animation to complete or the network request to finish, which in one test might just add up to seconds, but with a whole test suite could take minutes. And it would slow down development if we had to wait minutes to see if the change we just made broke anything.

So the first thing we should do when writing tests is create integration tests to cover our important use cases. Then we can look at the code coverage and fill in the holes with more integration tests or with unit tests. How many e2e tests we write depends on how much of a difference there is between the integration and e2e environments. For full-stack tests, there might be a lot of differences between the integration test runner and an actual browser, so we should at least test the critical path (the most important user actions, for example in [twitter.com](#)'s case, logging in, posting a tweet, and scrolling the feed). For backend, where the integration tests include apollo server's request pipeline, there's not much difference between integration and e2e—in which case we can just do a couple tests that make sure the HTTP server runs and the connection to the database works.

## Continuous integration

While continuous integration (CI) technically means merging to master frequently, in modern web development it usually means the process of tests being run automatically on each commit. It's often done with a service like [CircleCI](#) that monitors our commits on GitHub, runs the tests, and provides a webpage for each commit where we can view the test output. We can also set it up to do something after the test, such as:

- Mark a pull request as passing or failing the tests.
- Mark that commit as passing or failing by adding a red X or green checkmark next to the commit in the repository's history.
- If successful, deploy the code to a server—for example the staging or production server.

When the last step is included, the process may also be called continuous delivery or continuous deployment.

## Authentication

### Tokens vs. sessions

There are two main ways in which a server can verify that a client is a certain user: signed tokens and sessions.

A **signed token** is piece of data that is *cryptographically signed*—which means we can mathematically verify who wrote the data. When the data is a user ID, for example `123`, and the signer is someone we trust (either our server, or a trusted third-party server when we're using an authentication service like Auth0), then we can verify the signature and know that the client is user `123`. The most common type of signed token is a JWT, or [JSON Web Token](#).

A **session** is a period of time during which a certain client is considered logged in as a particular user. The server stores data about the session, for instance:

```
{
  sessionId: 'abc',
  userId: 123,
  expiresAt: 1595627896095
}
```

And gives the client a secret: in this case, the `sessionId`. Whenever the client contacts the server, the client includes the secret so that the server can look up the session data. For instance, if the client sends `'abc'`, the server can look up the above record, and if the session hasn't expired, the server knows the client is user `123`.

Both methods can contain additional information about the user—information commonly included in order to prevent the server from having to take the time to look up the user record from the database. For example, we could include authorization info like `isAdmin` or profile info like `name` and `email`.

There are some pros and cons to each method:

- **State:** Sessions are stateful—the server has to record the session data somewhere (in Redis, or in memory with sticky sessions), and that introduces complexity (and increased [latency](#) in the case of Redis). Signed tokens are stateless—all the information that the server needs is contained in the token.
- **Invalidation:** When a session secret is compromised, we can invalidate that session by deleting it from the data store. When a token is compromised, we can't invalidate it—it's already been signed and will continue to be valid until the expiration. We'd have to add a list of invalid tokens—either in code and re-deploy, or in a data store—and add logic to check them.

The differences are small enough that for most applications, we recommend using whichever method is easier to build.

## localStorage vs. cookies

We can store session secrets and signed tokens in either `localStorage` or cookies, which have different pros and cons:

- **Size:** Cookies can't be larger than 4KB, and in some cases we might want to store more data than that in our token, in which case we'd need to use `localStorage`.
- **Flexibility:** Data you put in `localStorage` can be managed by client-side JavaScript and sent to any domain, whereas cookies can only be set by the server and can only be shared among subdomains.
- **XSS:** Cookies are set by the server and can be configured to not be accessible from client-side JS, so they can't be accessed by [XSS attacks](#). Data stored in `localStorage` is vulnerable to XSS because it can be read by any JS running on your page (from any source allowed by your [CSP](#)).
- **CSRF:** Cookies are vulnerable to [CSRF attacks](#), whereas `localStorage` is not.

While the XSS issue is a serious concern, a common mitigation is setting short expirations, and for applications without strict security requirements, we again recommend using whichever method is easier to set up.

## Browser performance

Users notice when sites are slow, and they don't like it 😞. So if we want our users to feel good using our site, we want different things in the browser to happen at certain speeds.

First let's go over how the browser works. Because JavaScript is single-threaded, it can only run on a single CPU core. We can have particular pieces of JS run in [Web Workers](#), which can run on different cores, but most of our JS runs on one core, in the browser's **main thread**. The browser also needs to do most of its page **rendering** (parsing HTML and CSS, laying out elements, painting pixels into images, etc) in the main thread.

**Composition**, in which the pixel images are positioned, happens on the GPU.

A CPU core has a limited speed—it can only do a certain amount of work each millisecond. And because both JS and rendering happen on the same core, every millisecond our JS takes up is another millisecond the browser rendering has to wait before it can run. And the user won't see the page update until the browser has a chance to render.

Now that we know what's going on, let's think about different situations the user is in and how fast our site should be in each:

- **Page load:** The faster the better, but good targets are under 5 seconds *time to interactive* (the page is interactive when content has been displayed and the page is interactable—it can be scrolled, things can be clicked on, etc.) for the first visit and under 2 seconds for subsequent visits.
- **Response:** When humans take an action like clicking a button, and the page changes within 100 milliseconds, they generally perceive the response as immediate. If the response takes over 100ms, humans perceive a delay. If our click event handler runs code that takes 100ms on slow devices, then we want to break the code into two pieces: the minimum amount that will trigger the desired UI change, and the rest. And we schedule the rest to be done later:

```
button.onclick = () => {  
  updateUI()  
  window.requestIdleCallback(doTheRest)  
}
```

or in React:

```
class Foo extends Component {  
  onClick = () => {  
    this.setState({ something: 'different' })  
    window.requestIdleCallback(this.doTheRest)  
  }  
}
```

`requestIdleCallback()` runs the given function when the browser is idle, after it has finished rendering the changes triggered by `updateUI()` / `this.setState()`.

- **Animation:** Humans perceive a motion as smooth at 60 fps—when 60 frames are rendered per second. If we take 1000 milliseconds and divide by 60, we get 16. So while something is moving on the page, we want the browser to be able to render every 16ms. The browser needs 6ms to paint, which gives us 10ms left to run JS in. “Something moving” includes visual animations like entrances/exits and loading indicators, scrolling, and dragging.

# Chapter 1: Understanding GraphQL Through REST

Chapter 1 is a basic introduction to GraphQL through the eyes of the technology it's replacing, REST. If you're already sold on GraphQL and familiar with the basics, feel free to skip ahead to [Chapter 2: Query Language](#) for theory or [Chapter 5: Client Dev](#) and [Chapter 11: Server Dev](#) for coding.

Chapter contents:

- [Introduction](#)
  - [GraphQL as an alternative to a REST API](#)
  - [A simple REST API server](#)
  - [A simple GraphQL server](#)
  - [Querying a set of data](#)
  - [Filtering the data](#)
  - [Async data loading](#)
  - [Multiple types of data](#)
  - [Security & error handling](#)
  - [Tying this all together](#)
- 

## Introduction

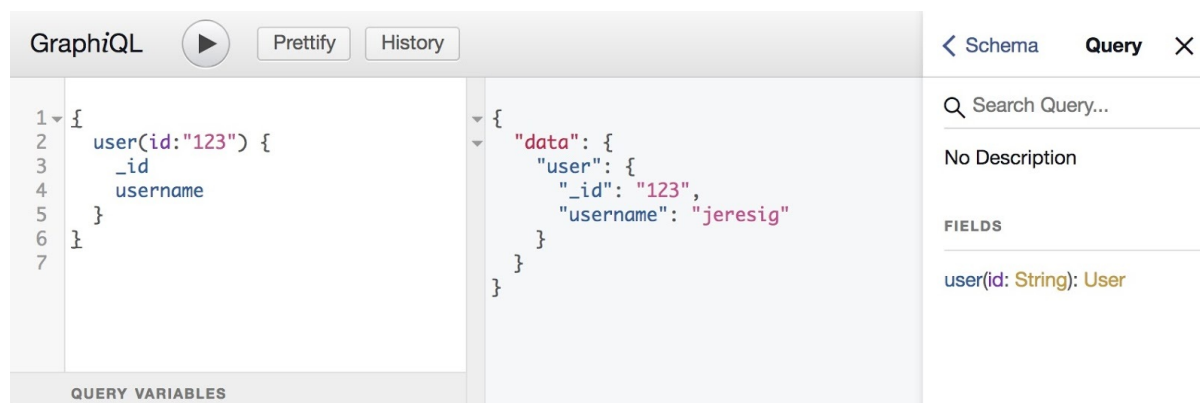
GraphQL is a modern, flexible specification for querying and modifying data that is on track to eclipse REST as a best practice for API design. You can write a single elegant query for all the data you need—no more cobbling together responses from a handful of REST endpoints.

When describing GraphQL, it's easier to start by saying what it isn't. It isn't a database—nothing is stored in it. It has little to do with graphs. While it is especially popular in the Node.js and React/Redux communities, GraphQL is a specification, which means it can be used in any language or framework. GraphQL is a query language (hence the “QL”)—it's an abstraction for querying and modifying data, and it's typically used by the client to fetch data from the server. It's an explicit, type-safe, and flexible alternative to traditional JSON REST APIs.

Why use GraphQL? It turns out that GraphQL is amazingly useful—it combines a number of features that have existed in well-designed REST APIs and presents them as a single, easy-to-understand package:

- **Flexible, explicit queries:** GraphQL puts the consumer of the API in full control over what data they receive. Instead of a REST endpoint that returns all the properties you could possibly want (and often links to more), you only get the properties you ask for.
- **Type-safe, self-documenting API:** GraphQL APIs are type-safe and self-documenting: the schema you define is exposed as interactive documentation for private or public consumption.
- **No more API endpoint sprawl:** Backend engineers also love GraphQL. Once you write the code for accessing a data type, you won't have to re-implement it. You don't have to make a new endpoint for each view—you can leave that work up to the client and the GraphQL execution model, which is implemented by your GraphQL server library. And you don't have to make a new API for each new app—you can have a single GraphQL API that covers all your business data.
- **Query consolidation:** A request for multiple data types can be combined into a single query that is executed in parallel on the server.
- **Static query analysis:** GraphQL schemas allow you to statically analyze the queries in your codebase, and they guarantee that you'll never break them.

All together you end up with an API that is a delight to use and allows you to write expressive queries with understandable results:



A GraphQL query (left) and the results from the server (right) inside the GraphiQL (with an “i” and pronounced “graphical”) web interface, an easy-to-use IDE for developing GraphQL queries.

By reading this book, you’ll learn why you should use GraphQL in production and how to do it, and some best practices and common pitfalls. To start, we’ll dip our toes in the water: we’ll see the power of GraphQL through examples and answer the most obvious questions. Then we’ll dive deep into the intricacies of how this system was designed.

## GraphQL as an alternative to a REST API

This chapter examines how to build a fully-functional API using REST techniques and how that compares to building an equivalent API using GraphQL.

A basic REST API (where we perform a HTTP GET request to a particular URL and get back some JSON data) is one of the simplest API designs in existence. The complications appear when we want to have greater control over the results returned from the server. This is where GraphQL’s abilities really shine.

While a GraphQL API starts out more complex than a REST API, its complexity doesn’t increase as quickly because GraphQL implementations implicitly handle many of the challenging aspects of API design. Once we understand the basics of GraphQL, we understand enough to do pretty much anything we want, which is an exciting proposition. On the other hand, while a REST API starts off simple, it quickly ratchets up in complexity to levels that can be challenging to maintain, as we’ll see later in this chapter.

GraphQL’s developers have taken all the best practices of an excellently designed REST API and built them into a single system. It may seem like there is a lot of abstraction to it, but that abstraction guides us toward practices that will make our APIs better designed and more consistent.

Going through building a REST API may seem like unnecessary work, but if we look at GraphQL as a REST API taken to its logical conclusion (a REST API with all the bells and whistles included), then a lot of the decisions that were made in designing GraphQL make a lot of sense. GraphQL is truly the best version of a REST API.

We recommend following along with the code, which can be found [on GitHub](#).

## A simple REST API server

Background: [Node](#), [MongoDB](#), [HTTP](#), [JSON](#)

We'll start our process of understanding GraphQL by building a simple REST API using Node.js and the popular Express web framework. We'll be retrieving data from a MongoDB database and using Mongoose as our object-relational mapping (ORM) to simplify querying the data we have stored.

Learn about MongoDB, the Node driver, and Mongoose in [Background > MongoDB](#).

In this application, our server will listen for requests to the `/users/:id` URL. We use the ID passed as a parameter in the URL (as specified by the `:id`) to query a user record from the database and return it as a JSON string. If we encounter any errors, we return a 500 error, and if we can't find the user, we return a 404—all standard REST practices.

`rest-server.js` :

```
const express = require('express')
const server = express()

// Get the Mongoose models used for querying the database
const { User } = require('./models.js')

// Listen for all GET requests to /users/:id URL (where the
// ID is the ID of the user account)
server.get('/users/:id', (req, res) => {
  // Try to find the user by their id (_id field), using the ID
  // parameter from the URL.
  User.findById(req.params.id, (err, user) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    if (!user) {
      // No user was found so we return a 404 error
      return res.status(404).end()
    }

    // Return the user to the client (automatically serialized
    // as a JSON string)
    res.send(user)
  })
})

// Start the application, listening on port 3000
server.listen(3000)
```

The Mongoose data models are stored in a separate file:

`models.js`

```
const mongoose = require('mongoose')

// Connect to the local MongoDB database named "testdb"
mongoose.connect('mongodb://localhost/testdb')

// Create a User schema to be stored in the MongoDB database
const UserSchema = new mongoose.Schema({
  _id: String,
  username: String
})

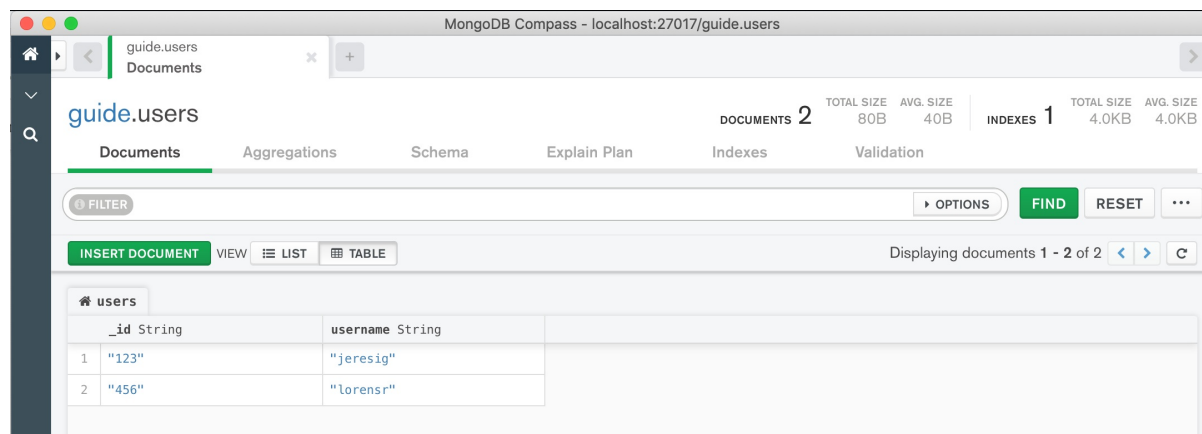
// Turn that schema into a model that we can query
const User = mongoose.model('User', UserSchema)

module.exports = { User }
```



We connect to the database and implement a schema for the `User` that has two fields, `_id` (the default ID field for MongoDB) and `username`, and we turn that schema into a model that lets Mongoose know that users should be stored in the `users` collection (it takes the given model name `'User'` and lowercases and pluralizes).

We'll need to have some data in our database to start, so we'll insert a couple of simple documents with string `_id` and `username` fields, looking something like this (in [MongoDB Compass](#), a desktop graphical interface for running MongoDB queries against a database):



The two records stored in the “users” collection of the MongoDB database.

To query this endpoint would be quite simple—we can run `curl` from the command line to verify that the endpoint's response matches our expectations:

```
$ curl http://localhost:3000/users/123
{"_id":"123","username":"jeresig"}

$ curl -I http://localhost:3000/users/abc
HTTP/1.1 404 Not Found
X-Powered-By: Express
Date: Sat, 02 Dec 2017 19:11:52 GMT
Connection: keep-alive
```

Querying for a user by their ID returns the expected JSON object, and if we try to find a user that's not in the database, we get the expected 404 error. Perfect!

## A simple GraphQL server

What does our REST server look like in the world of GraphQL? GraphQL has the concept of a schema—it's similar to those in Mongoose and other data model libraries, but a GraphQL schema is used differently. In Mongoose the schema is a representation of the data that's stored in the MongoDB database, but that's not necessarily the case for a GraphQL schema. A GraphQL schema doesn't need to match the storage format and can represent data from more than one source. To represent our basic `User`:

```
type User {
  _id: String!
  username: String!
}
```

This tells GraphQL that we have a type named `User` with two fields: `_id` and `username`, both strings. This alone doesn't really do anything, though. GraphQL doesn't know how to fetch this data or what interface(s) to set up for the client to query. We'll write a function for the former in a bit, and we can do the latter—define a simple query interface—using the same schema syntax as above:

```
type Query {
  user(id: String!): User
}
```

In this case, we're saying that we want a single query field (named `user`) that accepts a single argument (a required string named `id`) and returns a `User` type. This tells GraphQL how we want the client to be able to interact with the data, but GraphQL still doesn't know how to actually get that data out of our database. Thankfully, we can make good use of the Mongoose models that we built before. GraphQL just needs to know what to do when the client sends a `user(id)` query. The Node.js GraphQL implementation makes smart use of Promises—we only need to return a Promise that resolves to a `User`, like this:

```
function user({ id }) {
  return User.findById(id)
}
```

Putting it all together, we end up with a full GraphQL server:

`graphql-server.js`

```
const graphqlHTTP = require('express-graphql')
const { buildSchema } = require('graphql')
const express = require('express')
const server = express()

// Get the Mongoose models used for querying the database
const { User } = require('./models.js')

// Start up a GraphQL endpoint listening at /graphql
server.use(
  '/graphql',
  graphqlHTTP({
    // We construct our GraphQL schema which has two types:
    // The User type and the Query type (through which all
    // queries for data are defined)
    schema: buildSchema(`
      type User {
        _id: String
        username: String
      }

      type Query {
        user(id: String!): User
      }
    `),
    // The methods that we'll use to get the data for our
    // main queries
    rootValue: {
      // Get a user based on the ID and return it as a Promise
      user({ id }) {
        return User.findById(id)
      }
    },
    // Display the GraphQL web interface (for easy usage!)
    graphql: true
  })
)

// Start the application, listening on port 3000
server.listen(3000)
```

As before, this program creates an Express server, but instead of making a `/users/:id` endpoint, it sets up an endpoint at `/graphql` that, using the schema we provide, allows clients to make the `user(id)` query.

We can see now how an API consumer makes their query:

```
query {
```

```

user(id: "123") {
  _id
  username
}

```

In addition to having a custom language for specifying the schema, GraphQL also has a language for specifying queries. This is more verbose than a REST API: with REST, the query is embedded in the URL itself; in GraphQL, we specify the endpoint that we're calling ( `user` ) along with the argument ( `id` with a value of `"123"` ), and we also list every `user` field we want the server to return. This extra syntax is what makes GraphQL so flexible and explicit: it gives us an exact list of the data we are attempting to fetch.

We have a couple of options if we want to run this query and get the data back from the server. To start, let's use the command line to show how a typical query might be executed:

```

$ curl -X POST -H "Content-Type:application/json" \
> -d '{"query": "{user(id: \"123\") {_id username}}"}' \
> http://localhost:3000/graphql
{"data":{"user":{"_id":"123","username":"jeresig"}}}

```

We're submitting our query as a POST request to the GraphQL endpoint and getting back a JSON response, like with the REST API. The response format is a bit different—our data is returned inside the `"data"` property, and the structure of the data matches our GraphQL query.

What happens when we attempt to query for a user that doesn't exist? Does it return a 404 like with a REST API?

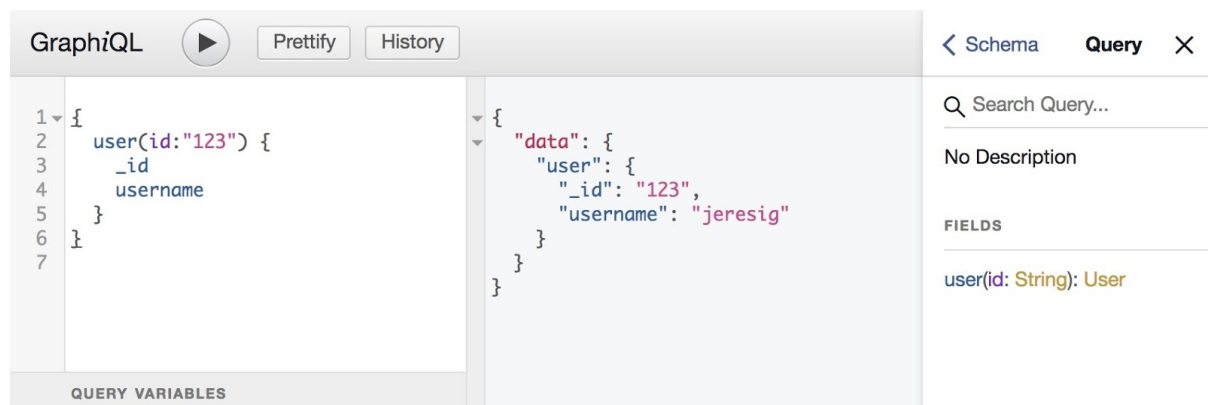
```

$ curl -X POST -H "Content-Type:application/json" \
> -d '{"query": "{user(id: \"123\") {_id username}}"}' \
> http://localhost:3000/graphql
{"data":{"user":null}}

```

No, in fact! Every time we query a GraphQL endpoint, we get a valid JSON response. In this case `"user"` is `null`, as its value wasn't able to be determined. This becomes very useful when we handle permissions and errors, which we'll get into [later](#).

When we run our server ( `node graphql-server.js` ) and visit [localhost:3000/graphql](http://localhost:3000/graphql) in the browser, we see GraphiQL::



*The results returned from a query inside the GraphiQL web interface.*

You can also try out a hosted version here: [ch1.graphql.guide/graphql](https://ch1.graphql.guide/graphql)

Seeing or using GraphiQL for the first time is often the moment that software engineers become GraphQL converts. It offers an intuitive interface to read the documentation for and query a GraphQL schema. We can write a query, see the results returned from the server, and explore the documentation on the right that's been automatically generated

from the schema. This is something that REST can't replicate without a ton of extra work or an additional framework. GraphQL is only just starting to pay off, though—as the REST API becomes more and more complex, the complexity of the equivalent GraphQL API remains the same.

## Querying a set of data

If we want to expand our REST API to allow for querying all of the users in our database, we need to add a new endpoint:

`rest-server.js` :

```
// Listen for all GET requests to /users
server.get('/users', (req, res) => {
  // Find all of the users in the database collection (we pass in
  // an empty collection as we aren't filtering the results)
  User.find({}, (err, users) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    // Return the array of users to the client (automatically
    // serialized as a JSON string)
    res.send(users)
  })
})
```

Like before, we can do a GET request to the new `/users` endpoint to see the user data returned as an array of objects:

```
$ curl http://localhost:3000/users
[{"_id":"123","username":"jeresig"}, {"_id":"456","username":"lorensr"}]
```

With our GraphQL endpoint, we can achieve a similar result by adding a `users` query to our schema:

```
type Query {
  user(id: String!): User
  users: [User]
}
```

And by adding an associated loader for that data, which is just a single function:

`graphql-server.js`

```
// The query fields that we'll use to get the data for our
// main queries
rootValue: {
  user({ id }) { ... },
  // Get an array of users and return them as a Promise
  users() {
    return User.find({})
  }
},
```

And that's all we need! The query syntax changes ever so slightly to make this new `users` query, but the rest of it stays intact. We still ask for the fields on the `user` type that we want, but we do so with the same syntax, even though we're operating against a set of users (rather than a single object). The query:

```
query {
  users {
```

```

    _id
    username
  }
}

```

This is one of the beauties of GraphQL: it's designed to scale easily from a single object to multiple with little change in our code. The result is as we would expect—it's just an array of user objects on the `"users"` attribute.

```

{"data":{"users":[{"_id":"123","username":"jeresig"}, {"_id":"456","username":"lorensr"}]}}

```

Now that we have the basics out of the way, let's look at some of the advanced features of REST APIs that GraphQL makes trivial.

## Filtering the data

In most REST APIs we are implicitly asking the endpoint to return all data, completely unfiltered. This could result in a potentially large request being sent back to the user along with a number of time- or resource-intensive sub-queries being executed to load particular fields or child data. All together that means a slow response time, especially on mobile. Many large REST APIs will end up adding a process for filtering the fields returned. For example, if we pass in a query string to our REST API that was something like `?fields=username` then we'd expect that the returned object(s) would only include the `username` field. We can achieve this by writing a function to filter the fields:

`rest-server.js`

```

// Filter a user object based on the requested fields
const filterFields = (req, user) => {
  const { fields } = req.query

  // If no fields were specified we return all of them
  if (!fields) {
    return user
  }

  // Otherwise we assume the fields are a comma-separated
  // list of field names, and we generate a new object that
  // contains only those fields.
  const filteredUser = {}
  for (const field of fields.split(',')) {
    filteredUser[field] = user[field]
  }
  return filteredUser
}

```

And then we need to ensure that every time we send a user object, we filter it to only contain the fields requested by the client. Note the altered `res.send()` lines at the end of each endpoint's handler function:

```

// Listen for all GET requests to /users/:id URL (where the
// ID is the ID of the user account)
server.get('/users/:id', (req, res) => {
  // Try to find the user by their id (_id field), using the ID
  // parameter from the URL.
  User.findById(req.params.id, (err, user) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    if (!user) {
      // No user was found so we return a 404 error
      return res.status(404).end()
    }
  })
})

```

```

    // Return the user to the client (automatically serialized
    // as a JSON string)
    res.send(filterFields(req, user))
  })
})

// Listen for all GET requests to /users
server.get('/users', (req, res) => {
  // Find all of the users in the database collection (we pass in
  // an empty collection as we aren't filtering the results)
  User.find({}, (err, users) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    // Return the array of users to the client (automatically
    // serialized as a JSON string)
    res.send(users.map(user => filterFields(req, user)))
  })
})

```

We can test to ensure it's working when querying a single user:

```

$ curl http://localhost:3000/users/123
{"_id":"123","username":"jeresig"}

$ curl http://localhost:3000/users/123?fields=username
{"username":"jeresig"}

$ curl http://localhost:3000/users/123?fields=_id,username
{"_id":"123","username":"jeresig"}

```

And also when querying all users:

```

$ curl http://localhost:3000/users
[{"_id":"123","username":"jeresig"}, {"_id":"456","username":"lorensr"}]

$ curl http://localhost:3000/users?fields=username
[{"username":"jeresig"}, {"username":"lorensr"}]

$ curl http://localhost:3000/users?fields=_id,username
[{"_id":"123","username":"jeresig"}, {"_id":"456","username":"lorensr"}]

```

With GraphQL, filtering is available by default. Remember how we had to specify which user fields we wished returned? GraphQL effectively requires that we specify a “fields” filter for every object. If we wanted to just fetch the username fields with GraphQL, the query would look like this:

```

query {
  users {
    username
  }
}

```

And the response would only include the fields that were specified:

```

{"data":{"users":[{"username":"jeresig"}, {"username":"lorensr"}]}}

```

Our field-filtering example is trivial: an object without any child objects. What would happen if the user object had a child object that also had fields we wished to include or exclude? What if some excluded fields took extra time to fetch from the database, and instead of just filtering them out, we wanted to avoid fetching them in the first place? The implementation of these things in the REST API sounds quite intimidating, so we'll leave that as an exercise to the reader 😊. With GraphQL, it's just a matter of specifying the fields we wish to include in our query. Having a standard

method of field specification means that it's easier for the server to avoid loading or querying unnecessary data from the database, and we can track precisely which fields are being used and which aren't. This is exciting, as we can use the field usage information to improve our database or help with migrating to a new schema. All of these benefits will be discussed in depth in this book.

## Async data loading

The data-loading code we've written so far has a simplistic structure: all of the data is held directly by the model. That's easy to manage in a REST API, but it gets harder when we want to return subobjects. For example, if each user were in a group, and we wanted that group's object to be returned along with the user, the code would become much more complex. Let's see what that would look like.

We need to update the `users` collection with a new `groupId` field:

🏠 users			
	<code>_id</code> String	<code>username</code> String	<code>groupId</code> String
1	"123"	"jeresig"	"dev"
2	"456"	"lorensr"	"author"

The updated user models with a new `groupId` field in the MongoDB database.

`groupId` refers to a group in the `groups` collection. A group has `_id` and `name` fields:

🏠 groups		
	<code>_id</code> String	<code>name</code> String
1	"author"	"Authors"
2	"dev"	"Developers"

The new group models in the MongoDB database.

We'd like to have the group object available as a property on the User model instead of the `groupId`:

```
$ curl http://localhost:3000/users/123
{"_id":"123","username":"jeresig","group":{"_id":"dev","name":"Developers"}}
```

Our first coding step is creating a Group model (which we'll use for both the REST and GraphQL implementations) to hold the Group details, and then we'll add a method to the User model for retrieving its associated Group (returning a Promise that resolves to that group).

`models.js`:

```
// Create a Group schema to be stored in the MongoDB database
const GroupSchema = new mongoose.Schema({
  _id: String,
  name: String
})

// Turn that schema into a model that we can query
const Group = mongoose.model('Group', GroupSchema)

// Create a User schema to be stored in the MongoDB database
```

```
const UserSchema = new mongoose.Schema({
  _id: String,
  username: String,
  groupId: String
})

// Retrieve the group associated with the user
UserSchema.methods.group = function() {
  // Use .exec() to ensure a true Promise is returned
  return Group.findById(this.groupId).exec()
}

// Turn that schema into a model that we can query
const User = mongoose.model('User', UserSchema)

module.exports = { User, Group }
```

We don't want to ever return the `groupId` field—instead we just want to return the group object (which can only be obtained by resolving the Promise returned from the `.group()` method). We'll need to update our application code in a number of ways to handle all of this asynchronous data loading. To start, we can update `filterFields()` to work asynchronously and resolve the Promises if they exist:

`rest-server.js`

```
// A list of the fields that are allowed to be accessed
const defaultFields = ['_id', 'username', 'group']

// Filter a user object based on the requested fields
const filterFields = async function(req, user) {
  // We assume the fields are a comma-separated list of field
  // names, if none is specified then we return all fields.
  const fieldKeys = req.query.fields
    ? req.query.fields.split(',')
    : defaultFields

  // Generate a new object that contains only those fields.
  const filteredUser = {}
  for (const field of fieldKeys) {
    // If the field is a function then we expect it to return
    // a Promise which we will immediately resolve.
    if (typeof user[field] === 'function') {
      filteredUser[field] = await user[field]()
    } else {
      filteredUser[field] = user[field]
    }
  }
  return filteredUser
}
```

We have to add a list of `defaultFields`, as we want to ensure that `group` is included and `groupId` is excluded. Now we can use our new asynchronous function in our API endpoints. For the first, we just make the `findById` callback `async` and `await filterFields()` before sending the response. For the second, we have to use `Promise.all()`:

```
// Listen for all GET requests to /users/:id URL (where the
// ID is the ID of the user account)
server.get('/users/:id', (req, res) => {
  // Try to find the user by their id (_id field), using the ID
  // parameter from the URL.
  User.findById(req.params.id, async (err, user) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    if (!user) {
      // No user was found so we return a 404 error
      return res.status(404).end()
    }
  })
})
```



```

    // Return the user to the client (automatically serialized
    // as a JSON string). We need to wait for all of the fields
    // to load before we can return the results.
    res.send(await filterFields(req, user))
  })
})

// Listen for all GET requests to /users
server.get('/users', (req, res) => {
  // Find all of the users in the database collection (we pass in
  // an empty collection as we aren't filtering the results)
  User.find({}, async (err, users) => {
    if (err) {
      // The DB returned an error so we return a 500 error
      return res.status(500).end()
    }

    // Return the array of users to the client (automatically
    // serialized as a JSON string) We need to wait for all
    // of the Promises to resolve for all of the users.
    res.send(await Promise.all(users.map(user => filterFields(req, user))))
  })
})

```

This solution works exactly as we expect it to, returning a group object along with the other fields:

```

$ curl http://localhost:3000/users/123
{"_id":"123","username":"jeresig","group":{"_id":"dev","name":"Developers"}}

$ curl http://localhost:3000/users/123?fields=username,group
{"username":"jeresig","group":{"_id":"dev","name":"Developers"}}

$ curl http://localhost:3000/users
[{"_id":"123","username":"jeresig","group":{"_id":"dev","name":"Developers"}}, {"_id":"456","username":"lorensr",
,"group":{"_id":"author","name":"Authors"}}]

$ curl http://localhost:3000/users?fields=username,group
[{"username":"jeresig","group":{"_id":"dev","name":"Developers"}}, {"username":"lorensr","group":{"_id":"author",
"name":"Authors"}}]

```

The code has become harder to follow and understand—there's no longer a clean one-to-one relationship between the data on the model and what we want to return, and handling asynchronous functions ramped up the complexity.

Let's compare this solution with how it would work in GraphQL. We already have the changes to the Mongoose models, so we start out by updating the GraphQL schema to represent the new Group type:

```

type Group {
  _id: String!
  name: String!
}

type User {
  _id: String!
  username: String!
  group: Group
}

```

And... that's it. That's all the work that we need to do (beyond the minor changes that were made to the Mongoose models). We can open GraphiQL to try out our new field and see that it works immediately:

The screenshot shows the GraphQL web interface. On the left, a query is defined: `query { users { _id username group { name } } }`. On the right, the results are displayed in a JSON format: `{ "data": { "users": [ { "_id": "123", "username": "jeresig", "group": { "name": "Developers" } }, { "_id": "456", "username": "lorensr", "group": { "name": "Authors" } } ] } }`. The interface also includes a 'Schema' tab, a search bar, and a list of fields: `user(id: String!): User`, `users: [User]`, `group(id: String!): Group`, and `groups: [Group]`.

*The results for all users and their groups inside the GraphQL web interface.*

GraphQL automatically handles values that are returned as Promises. The GraphQL server attempted to resolve the `group` field by executing the User model's `.group` method and waiting until the Promise resolved before including the value.

Notice that because GraphQL requires filtering fields by specifying their names, we also have field filtering on the group submodel. We don't have this in our REST API implementation.

It's also important to note that GraphQL follows a best practice: it doesn't query for any data that it doesn't need. If the user never explicitly requests the `group` field, then the server won't perform the database query to retrieve it. Following GraphQL's patterns will result in an API that's designed correctly—and optimally—from the get-go.

As the complexity of our data model starts to increase, so does the complexity of the implementation of our REST API (which is, due to all the features we've added to it, arguably already approaching unmaintainable levels of complexity). In contrast, GraphQL scales very gracefully: multiple models are no more challenging than one, and asynchronous data is just as easy as synchronous.

## Multiple types of data

Thus far we've been retrieving data in a way that is stylistically similar to the REST API endpoint: we request a single user, or a list of users, and that's it. Retrieving data in this way aligns well with the expectations of a normal, well-designed REST API. However, fundamentally, a REST API is not designed for the data requirements of a modern application. Modern applications need to access many different types of data simultaneously in order to successfully render a result. An app might need to show not just a User but also their Posts and the Comments.

The client should be in control of requesting the data they want from the server. Ideally, this should be done in a single HTTP request. With REST, we can either make multiple HTTP requests to get the different types of data we need for a page, or we can design a custom REST endpoint that returns everything all at once. GraphQL employs a different strategy: a GraphQL endpoint can return many types and execute many queries, not just one. This gives the caller the power of being able to fetch any and all data needed in a single request.

If we want to request multiple types of data with a basic REST API—one in which each endpoint deals with a single type—our client code might look something like this:

```
const getUserWithGroup = user =>
  fetch(`http://localhost:3000/groups/${user.groupId}`)
    .then(response => response.json())
    .then(group => ({
      ...user,
      group
    }));
```

```

    )))

    fetch('http://localhost:3000/users')
    .then(response => response.json())
    .then(users => Promise.all(users.map(getUserWithGroup)))
    .then(usersWithGroups => {
      console.log(usersWithGroups)
    })
  })
}

```

First we request the users from the server, and once they've been returned, we request each user's group. Once all the groups are returned, we have an array of full user objects to use. Here is the equivalent code using our GraphQL API:

```

import { request } from 'graphql-request'

const query = `{
  users {
    username
    group {
      name
    }
  }
}`

request('http://localhost:3000/graphql', query).then(({ users }) => {
  console.log(users)
})

```

We don't have to wait for two round-trip requests to the server, and we don't have to write code to manage the Promises or combine the data in the response objects.

As the complexity of a query scales, the conciseness of GraphQL becomes increasingly compelling. Let's say we wanted to get the current `User`, their `Post`s, and each post's `Comment`s:

```

import { request } from 'graphql-request'

const query = `{
  currentUser {
    username
    posts {
      title
      comments {
        text
        createdAt
      }
    }
  }
}`

request('http://localhost:3000/graphql', query).then(({ currentUser }) => {
  console.log(currentUser)
})

```

We get back a user object that has a list of posts, and each post has a list of comments. The equivalent REST logic is even more complicated than our `usersWithGroups` example, the latency would increase to three round trips (first for the user, then for their posts, and then once we have the posts, for the posts' comments), and the total number of requests would be very high. For example, if the user had 5 posts, each of which had 4 comments, we'd be sending 26 requests: 1 to get the user, 5 to get the posts, and  $5 * 4 = 20$  to get all the comments.

We could simplify the REST client code and reduce latency by adding more complexity to our REST API. Instead of our endpoints dealing with a single type, we could have them return multiple types, as we did for the user's `group` field in the last section. We could also get all the current user's post and comment data with

`http://localhost:3000/currentUser?fields=posts,posts_comments` , but it would require more complex logic for filtering nested fields.

We've looked at the differences between using a REST API and a GraphQL API to fetch a single type that has fields of other types, but what about fetching multiple different top-level types? If we're implementing an app's homepage, we might want the current user's name and photo, a list of their recent notifications, and a list of the most recent posts. We could fetch that in three requests with `/currentUser` , `/notifications` , and `/posts` , but if we wanted to fetch all the data in a single request, we would need a view-specific endpoint, for example `/homepage` . There are a few issues with building and maintaining view-specific endpoints. Between all our client platforms—for example, web, iOS, and Android—we could have a lot of views, which means a lot of endpoints to code. Even if the endpoints are all using the same model layer for database access, there's still the logic of putting together the response object and supporting any query parameters we might want. When we remove a part of a view—for instance, the recent posts from the homepage—the client is overfetching (getting more data than it needs) until we update the `/homepage` endpoint. And when we want to add new parts to views, we need to wait for the backend team to add the required data to the endpoint. Add versioning to all these changing endpoints in order to keep supporting older native clients (or older developer integrations, in the case of public APIs), and we've got a huge mess.

Fetching multiple top-level types from a GraphQL API doesn't require that much new code. Let's say we wanted to get a list of all of the Users and Groups in our database in a single request. We have to add in the new access points to get the Group data, like we did for the User:

`graphql-server.js` :

```
// Get the Mongoose models used for querying the database
const {User, Group} = require('./models.js')

// Start up a GraphQL endpoint listening at /graphql
server.use(
  '/graphql',
  graphqlHTTP({
    schema: buildSchema(`
      ...

      type Query {
        user(id: String!): User
        users: [User]
        group(id: String!): Group
        groups: [Group]
      }
    `),
    // The query fields that we'll use to get the data for our
    // main queries
    rootValue: {
      user({id}) { ... },
      users() { ... },

      // Get a group based on the ID and return it as a Promise
      group({id}) {
        return Group.findById(id)
      },
      // Get an array of groups and return them as a Promise
      groups() {
        return Group.find({})
      }
    }
  })
)
```

And now it works! If we load up our GraphiQL web interface, we can see that we not only have access to the existing `user` and `users` query fields, but also to the new `group` and `groups` fields. More importantly, we can include multiple query fields in a single request:

The screenshot shows the GraphQL web interface with a query on the left and its JSON response on the right. The query is:

```

1 query {
2   users {
3     _id
4     username
5     group {
6       name
7     }
8   }
9   groups {
10    name
11  }
12 }
13

```

The response is:

```

{
  "data": {
    "users": [
      {
        "_id": "123",
        "username": "jeresig",
        "group": {
          "name": "Developers"
        }
      },
      {
        "_id": "456",
        "username": "lorensr",
        "group": {
          "name": "Authors"
        }
      }
    ],
    "groups": [
      {
        "name": "Developers"
      },
      {
        "name": "Authors"
      }
    ]
  }
}

```

On the right, the schema is visible with fields: `user(id: String!): User`, `users: [User]`, `group(id: String!): Group`, and `groups: [Group]`.

The results for all users and all groups inside the GraphQL web interface.

Here, we've retrieved a list of all the users along with the group they're in. Additionally, we've retrieved a complete list of all the groups in the database, and all of this information was retrieved in parallel.

What's especially nice is that we don't have to limit ourselves to just lists of data—we can mix in any number of query fields, like the `group` query field:

The screenshot shows the GraphQL web interface with a query on the left and its JSON response on the right. The query is:

```

1 query {
2   users {
3     _id
4     username
5     group {
6       name
7     }
8   }
9   groups {
10    name
11  }
12   group(id: "dev") {
13     name
14   }
15 }
16

```

The response is:

```

{
  "data": {
    "users": [
      {
        "_id": "123",
        "username": "jeresig",
        "group": {
          "name": "Developers"
        }
      },
      {
        "_id": "456",
        "username": "lorensr",
        "group": {
          "name": "Authors"
        }
      }
    ],
    "groups": [
      {
        "name": "Developers"
      },
      {
        "name": "Authors"
      }
    ],
    "group": {
      "name": "Developers"
    }
  }
}

```

On the right, the schema is visible with fields: `user(id: String!): User`, `users: [User]`, `group(id: String!): Group`, and `groups: [Group]`.

The results for all users, groups, and a single group inside the GraphQL web interface.

In this case, we're requesting three query fields simultaneously (getting a list of all users, a list of all groups, and also a specific group) and returning all the data in a single request. This represents a level of customization and flexibility that is quite challenging to implement with a traditional REST API.

In summary, the advantages of using GraphQL for fetching multiple data types are:

- **The client retains control over its data requirements:** Instead of the REST endpoint dictating the queries

being run and the data returned, the client can specify the queries and the desired data and get it all back in a single request.

- **Simpler server:** The server doesn't have to attempt to permute all of the possible desired endpoints. This helps reduce the cost and surface area for the API. We don't have to know about all the use cases or platforms that the data will eventually appear in, so the implementation becomes much simpler and easier to maintain.
- **Fewer requests:** It reduces the number of distinct requests for data, and thus the burden on both the client and the server. If we were to request three different pieces of data from a REST API, it could potentially require three different endpoints and three distinct HTTP requests. With GraphQL, we're guaranteed to have a single request and the same unchanged implementation.
- **Faster:** It reduces the latency in the overall request by allowing most of the data loading to be done on the server rather than the client (which has to wait for the current HTTP request to complete before initiating any other requests that depend on the current request's response).

## Security & error handling

When it comes to the security of our data (validating the permissions of those that are attempting to access it) and the handling of errors, REST APIs have an idiomatic solution: returning a specific error code. For example, if we attempt to access data which we don't have permission to access, we might get an HTTP 403 Forbidden code in response. If our request results in an error, then we might get an HTTP 500 Internal Server Error code. Some REST APIs might include detailed information on the failure inside the response body (such as the error message or the specific data that we don't have permission to access), but the error codes are generally used to designate the class of error, not the specific error itself.

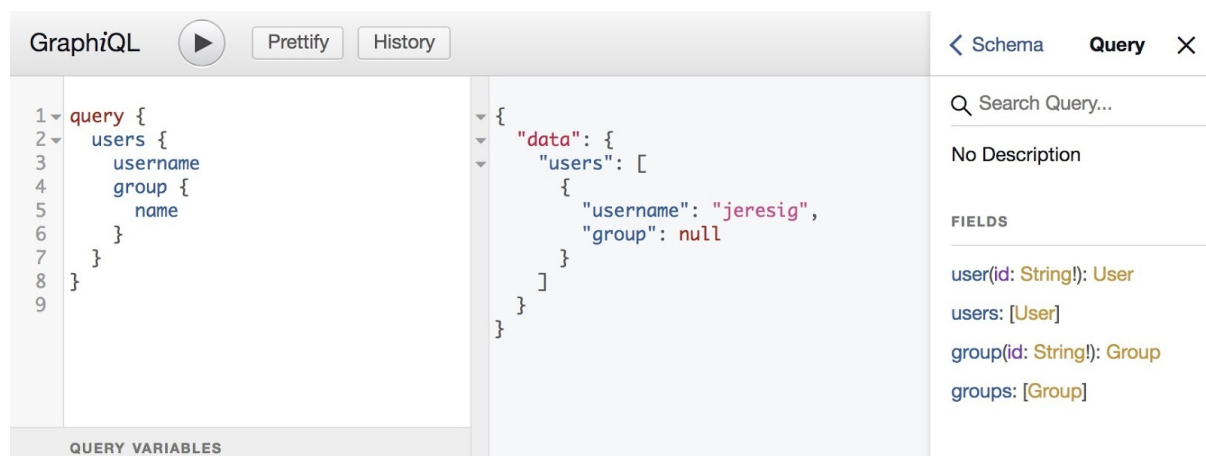
In GraphQL every request is expected to return a result, even if that result doesn't have the data we request. GraphQL tends to treat security and error-handling issues similarly. If there's a problem with accessing a specific piece of data, then `null` is returned in its place. This calls for a defensive, but smart, way of coding an application. Since no field is guaranteed to be there, we need to ensure that it exists before attempting to use it.

For example, let's say the MongoDB server wasn't running. The output from our server might look something like this:

```
{
  "errors": [
    {
      "message": "failed to reconnect after 30 attempts with interval 1000 ms",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "users"
      ]
    }
  ],
  "data": {
    "users": null
  }
}
```

In this case, the `"users"` property ends up being `null`, and there's an additional `"errors"` property that has a list of errors that were thrown, including where each error came from—the field name and the location in the query.

Whether a user doesn't have permission to access a piece of information or the data doesn't exist, the end result is similar: the user doesn't get that data and is given a `null` result instead, like in the following:



An example of a failed load for a group child object inside the GraphQL web interface.

In this case, we either didn't have an associated group or we didn't have permission to access that group, so we were given a `null` value. GraphQL gives the user as much data as it possibly can, leaving out anything that's missing (for one reason or other). This also allows the application to render a version of the site that has some portion of the interface available, and allows it to make note of missing information, rather than displaying a general "Error!" message that contains no context.

GraphQL's design provides a level of consistency that should be greatly appreciated by all developers. Every request will return a valid JSON response (unless something goes very wrong). There is no guarantee that the response will contain all of the data we request, so we end up building more resilience into our application. This should be a best practice, as it provides an optimal experience to the user.

## Tying this all together

REST APIs have served us well for many years. They've made data access for many applications easy to understand and implement. However, as we've seen in this chapter, this simplistic approach belies the true complexity of implementing a full API that supports a modern web or mobile application.

In a modern application, the consumer needs to be in control of what data they can request. A single page may have many different data models represented in it, and short of writing a unique REST endpoint for every page of a site, REST simply doesn't have the flexibility to allow applications to request the data they need at all times.

While GraphQL has a number of new concepts to learn (GraphQL schemas, the query language, etc.), these features are designed to help us write our applications correctly from the start—whereas the prospect of building a comparable REST API can be absolutely overwhelming in its complexity.

GraphQL truly is the most developer-friendly way of building an API. It puts the consumer in full control of the data requested, and we can therefore avoid querying data that we don't need. As an added bonus, there is clear, automatically-generated documentation we can browse to understand any new GraphQL API.

The developers of GraphQL learned from REST's challenges and mistakes over the years, and have turned the best parts into a streamlined interface that will surely be the standard for API design for many years to come. The rest of this book will dive deep into the benefits of GraphQL, how to implement it efficiently, and how to build the best applications using this technology.





## Chapter 2: Query Language

Chapters 2–4 are a complete reference for the GraphQL specification. If you want to get straight to coding, feel free to skip ahead to [Chapter 5: Client Dev](#) or [Chapter 11: Server Dev](#) and refer back here when necessary 😊. Or if you have the time and want a thorough base of understanding, take it in order.

Chapter contents:

- [Document](#)
- [Fields](#)
- [Arguments](#)
- [Fragments](#)
- [Variables](#)
- [Directives](#)
- [Mutations](#)
- [Subscriptions](#)

---

TODO write this chapter 🙄. For now, linking to [graphql.org](https://graphql.org)

## Document

Similar to how we call a JSON file or string a JSON *document*, a GraphQL query file or string is called a GraphQL *query document*. The [GraphQL spec](#) defines how to write a document:

A document is a list of one or more *operations* or [fragments](#). An operation is either a `query`, `mutation`, or `subscription`. Our beginning examples are documents with a single `query` operation, like this:

```
query {
  githubStars
}
```

Our operation has a single root query [field](#), `githubStars`. In this type of document—a single `query` operation without [variables](#) or [directives](#)—we can omit `query`, so the above document is equivalent to:

```
{
  githubStars
}
```

A more complex document could be:

```
query StarsAndChapter {
  githubStars
  chapter(id: 0) {
    title
  }
}

subscription StarsSubscription {
  githubStars
}

mutation ViewedJavascriptSection {
  viewedSection(id: "0-1") {
```

```
    ...SectionData
  }
}

mutation ViewedGitSection {
  viewedSection(id: "0-2") {
    ...SectionData
  }
}

fragment SectionData on Section {
  id
  title
}
```

It has all the operation types as well as a fragment. Note that when we have more than one type of operation, we need to give them names—in this case, `StarsAndChapter`, `ViewedDocumentSection`, and `StarsSubscription`.

The content between an operation's outer curly braces is called the *selection set*—the list of data fields we're requesting. For instance, the selection set of `StarsAndChapter` is:

```
{
  githubStars
  chapter(id: 0) {
    title
  }
}
```

We can think of the selection set as the left-hand side of a JSON document (with just the attributes), where the response will be the full JSON (with all the values filled in).

## Fields

[fields](#)

## Arguments

[arguments](#)

## Fragments

[fragments](#)

[inline fragments](#)

## Variables

[variables](#)

## Directives

[directives](#)

## Mutations

[mutations](#)

## Subscriptions

[subscriptions](#)

## Chapter 3: Type System

Chapter contents:

- [Schema](#)
- [Scalar types](#)
- [Enum types](#)
- [Object types](#)
- [Query & Mutation types](#)
- [Lists](#)
- [Non-null](#)
- [Arguments](#)
- [Unions](#)
- [Interfaces](#)

---

TODO write this chapter 😊. For now, linking to [graphql.org](https://graphql.org)

### Schema

[schema](#)

### Scalar types

[scalar types](#)

### Enum types

[enum types](#)

### Object types

[object types](#)

### Query & Mutation types

[Query & Mutation types](#)

### Lists

[lists](#)

## Non-null

[non-null](#)

## Arguments

[arguments](#)

[object arguments](#)

## Unions

[unions](#)

## Interfaces

[interfaces](#)

## Chapter 4: Validation & Execution

Chapter contents:

- [Validation](#)
- [Resolvers](#)
- [Execution](#)
- [Error handling](#)

---

TODO write this chapter 🤔. For now, linking to [graphql.org](https://graphql.org)

### Validation

[validation](#)

### Resolvers

[resolvers](#)

### Execution

[execution](#)

### Error handling

todo

## Chapter 5: Client Dev

This begins the practical coding part of the book. 🎉👨‍💻

Client-side:

- [Chapter 5: Client Dev](#)
- Web:
  - [Chapter 6: React](#)
  - [Chapter 7: Vue](#)
- Mobile:
  - [Chapter 8: React Native](#)
  - [Chapter 9: iOS](#)
  - [Chapter 10: Android](#)

Server-side:

- [Chapter 11: Server Dev](#)
- 

Chapter 5 contents:

- [Anywhere: HTTP](#)
    - [cURL](#)
    - [JavaScript](#)
  - [Client libraries](#)
    - [Streamlined request function](#)
    - [View layer integration](#)
    - [Caching](#)
    - [Typing](#)
    - [DevTools](#)
- 

GraphQL can be used between any two computers, such as from a web browser to a server or between two servers. Any computer with a network connection can send a GraphQL request, and any computer with an IP address on that network can receive that request and send back a response. Most software written these days follows the [client–server model](#), in which one computer is always providing a service (a **server**), and another computer is always requesting the service (a **client**, such as a web browser or mobile app). In a GraphQL client–server model, the client makes GraphQL requests, and the server provides the service of responding to those requests. We'll code GraphQL clients in the next few chapters and a GraphQL server in the [last](#).

First we'll make simple HTTP requests, which we can do from any computer. Most application clients are web browsers or mobile apps, so after HTTP, we'll use the best web and mobile GraphQL libraries to create full-featured clients. The two web view layers we'll be covering are React and Vue, and we'll use their most popular GraphQL libraries, which are [react-apollo](#) and [vue-apollo](#). The best mobile libraries are [Apollo iOS](#), [Apollo-Android](#), and [react-apollo](#) for React Native.

For each type of client and the server, the app we'll go through building is [graphql.guide](#)—a web or mobile app for reading the GraphQL Guide.

## Anywhere: HTTP

---

Background: [HTTP](#), [JSON](#)

Whether we're writing JavaScript for a website, Swift for an iPhone, C for a microcontroller, etc., we can make a connection to a server and send an [HTTP request](#). At its base, a GraphQL request is just an HTTP POST request.

## cURL

When we're on the command line, we can use [cURL](#) ("See URL", a tool for making network requests, including HTTP requests):

```
$ curl -X POST \
-H "Content-Type: application/json" \
-d '{"query": "{ githubStars }"}' \
https://api.graphql.guide/graphql
```

- `-x` specifies which HTTP method to use—in this case POST
- `\` continues the command on the next line
- `-H` sets an HTTP header—in this case the [Content-Type](#) header (where we specify the [MIME type](#) of the request body) to `application/json`
- `-d` sets the body of the request—in this case to our [JSON](#) query: `{"query": "{ githubStars }"}`

`curl` prints the response to the command line:

```
{"data":{"githubStars":1337}}
```

When talking about GraphQL, we usually skip over:

- the `{"query": "x"}` part of the request body
- the `{"data": y}` part of the JSON response

Instead we just talk about:

- `x`, the GraphQL [document](#): `{ githubStars }`
- `y`, the value of the `"data"` attribute: `{"githubStars":1337}`

## JavaScript

In a browser, we can use [fetch\(\)](#) to make HTTP requests:

```
const makeGraphQLRequest = async ({ endpoint, document }) => {
  const options = {
    method: 'POST',
    headers: new Headers({
      'Content-Type': 'application/json'
    }),
    body: JSON.stringify({
      query: document
    })
  }

  const response = await fetch(endpoint, options)
  return response.json()
}

const logStars = async () => {
  const response = await makeGraphQLRequest({
    endpoint: 'https://api.graphql.guide/graphql',
    document: '{ githubStars }'
  })
}
```



```
console.log(response)
}
```

### Run in browser

Just as we did with cURL, we make an HTTP POST request to our endpoint url with a Content-Type header and a JSON body. Running `logStars()` prints this to the console:

```
{
  data: {
    githubStars: 1337
  }
}
```

We can do the same thing in other languages by using their HTTP request functions with equivalent parameters.

For our in-browser JavaScript example, instead of logging the data, we can display it on the page:

```
const displayStars = async () => {
  const response = await makeGraphQLRequest({
    endpoint: 'https://api.graphql.guide/graphql',
    document: '{ githubStars }'
  })
  const starCount = response.data.githubStars
  const el = document.getElementById('github-stars')
  el.innerText = `The Guide has ${starCount} stars on GitHub!`
}

displayStars()
```

### Run in browser

This method of displaying data (finding a DOM node with `document.getElementById` or `document.querySelector` and setting its `innerText` ) is straightforward and great for simple tasks. However, most web apps we build are complex enough that they benefit greatly from a user interface library like React—in the [next chapter](#), we'll learn the best way to put GraphQL data into our React components.

## Client libraries

There are many different GraphQL client libraries for different platforms and languages. Here is some common functionality that the libraries might provide:

- [Streamlined request function](#)
- [Typing](#)
- [View layer integration](#)
- [Caching](#)
- [DevTools](#)

The first is useful anywhere—whether it's a script, service, or website that's mostly static (only displays a small amount of dynamically fetched data). The second is useful when we're working in a typed programming language. The last three are extremely helpful for building applications: whether we're making a web app, a mobile app, or a desktop app, we usually need to fetch and display a number of different types of data from the server, and decide which to fetch and display based on user interactions. We also want to remember what we requested in the past, because often when we need to display it again, we don't need to fetch it again. Doing all of this ourselves can get really complicated, but advanced client libraries like Apollo can take care of a lot of it for us.

## Streamlined request function

The most basic thing a library does is give us something like the `makeGraphQLRequest()` function we wrote above, which takes care of constructing the HTTP POST request and parsing the response. For instance, the `graphql-request` client library does this (and only this).

## Typing

When we're working in typed languages, we have to write our own object types and models, and when we get JSON from a REST API, we have to convert the data into our types. With GraphQL, we know the types for everything because they're in the schema. Which means that our client libraries can provide us with type definitions or generate typed model code for us. For instance, `apollo-codegen` generates type definitions for Typescript, Flow, and Scala, `Apollo iOS` returns query-specific Swift types, and `Apollo-Android` generates typed Java models based on our queries and schema.

A combination of having a schema and query documents also allow for some great code editor features, such as autocomplete, go to definition, and schema validation. (See, for example, the [VS Code plugin](#) and [IntelliJ/WebStorm plugin](#).)

## View layer integration

Most libraries even take care of calling `makeGraphQLRequest` for us: we simply add our query documents to the components that need them, and when those components are rendered, the documents get combined into a request to the server. When we get data back from a GraphQL server, we usually want to display it to the user—and depending on the view layer, our library may have handy ways of helping us do that, as well as handling loading state and errors. We'll see some specific examples of this in the [React](#), [React Native](#), and [Vue](#) sections.

## Caching

Background: [latency](#)

All of the libraries we'll use cache the data we get from the GraphQL server—that is, they store the data, either in memory or on disk, so that we can immediately access it later without having to request it again from the server. While some libraries just cache the whole response and give us the cached response when we make the exact same request, the most useful libraries store the data in a **normalized** cache. A normalized cache breaks down the response and saves each object separately, so that if we make a different, overlapping query, the library can still give us the cached data. As an example, let's consider this part of the schema of the Guide:

```
type Query {
  currentUser: User
  section(id: Int!): Section
}

type User {
  firstName: String
  hasRead: [Section]
}

type Section {
  title: String!
}
```

When a user visits their profile, we want to show a list of which sections they've read, so we send this query:

```

{
  currentUser {
    id
    firstName
    hasRead {
      id
      title
    }
  }
}

```

Our query returns:

```

{
  "currentUser": {
    "id": "1",
    "firstName": "Loren",
    "hasRead": [
      {
        "id": "5_1",
        "title": "Anywhere: HTTP"
      },
      {
        "id": "5_2",
        "title": "Client Libraries"
      }
    ]
  }
}

```

And our library saves the response in a normalized cache. The key to each object in the cache is a string of the format "[type]:[object id]" , for instance "User:1" for a User object with an id of 1:

```

cache = {
  "User:1": {
    id: "1",
    firstName: "Loren",
    hasRead: [
      "Section:5_1",
      "Section:5_2"
    ]
  },
  "Section:5_1": {
    id: "5_1",
    title: "Anywhere: HTTP"
  },
  "Section:5_2": {
    id: "5_2",
    title: "Client Libraries"
  }
}

```

If the user then navigates to section 5\_2 , we want to display the title of that section, so we look it up with this query:

```

{
  section(id: "5_2") {
    title
  }
}

```

Which should return:

```

{
  "section": {
    "title": "Client Libraries"
  }
}

```

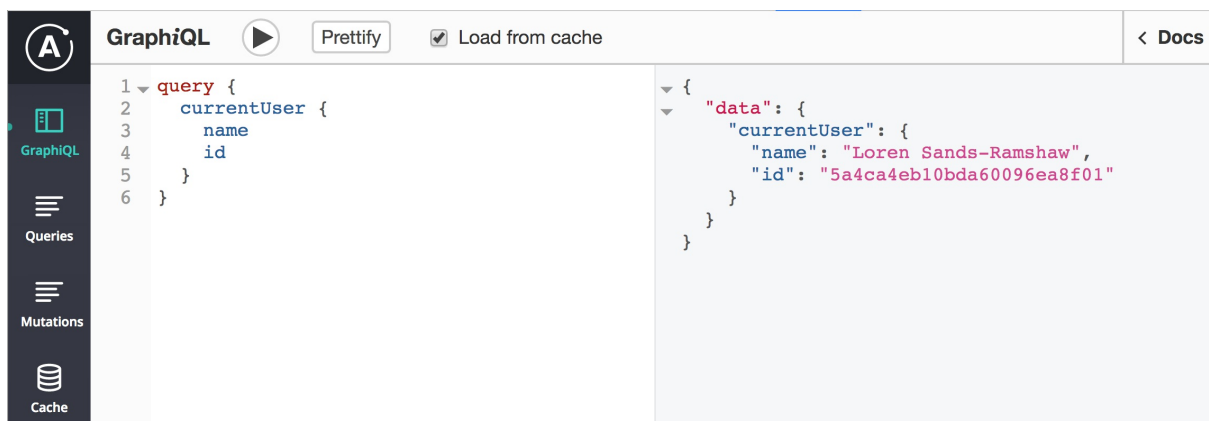
```
}
}
```

But we already know from our first query that `"Client Libraries"` is the title. Our normalized caching library, instead of sending our second query to the server, can find the object with the `"Section:5_2"` key in the cache and immediately return it to us.

Sometimes we want to re-request data from the server, because it may have changed since we originally requested it. Client libraries certainly allow us to re-request, but we usually want to immediately display the cached data while we wait for the response to arrive. The time it takes to get a value from the cache and display it on the screen could be as little as 20ms, whereas common response times from GraphQL servers are on the order of hundreds of milliseconds or seconds, depending on the user's network connection [latency](#) and how long the server takes to retrieve their data. Humans [perceive delays](#) above 100ms, so UX best practice is to show something on the screen within 100ms of a user action. We probably won't get a response from our GraphQL server in time, so we'll want to either communicate that we're waiting—for example by displaying a spinner—or display the cached version of the data.

## DevTools

Some libraries have browser DevTools extensions for viewing information about an app's GraphQL operations and the current state of the store. Apollo's DevTools also has a built-in GraphiQL that uses the same network link as our app, so we don't have to manually set authentication headers. We can even use it to query the store instead of the server by checking "Load from cache":



Under the Queries tab, we see the current "Watched queries" (queries attached to our components) and what variables they were called with:

Developer Tools - http://localhost:3000/Introduction

Elements Console Sources Network Performance Memory Apollo >>

**Watched queries**

1. UserQuery
2. StarsQuery
3. ChapterQuery
4. SectionContent

**SectionContent <Query>** Run in GraphQL

▼ **Variables**

id "intro"

▼ **Query string**

```
query SectionContent($id: String!) {
  section(id: $id) {
    id
    content
    views
    __typename
  }
}
```

Under the Mutations tab, we see a log of all past mutations and their variables:

**Mutation log**

1. ViewedSection

**ViewedSectionShow in GraphQL**

▼ **Variables**

id "intro"

▼ **Mutation string**

```
mutation ViewedSection($id: String!) {
  viewedSection(id: $id) {
    id
    views
    __typename
  }
}
```

Under the Cache tab, we see the current state of the store/cache. Normalized data objects (any objects for which the `id` was requested) are listed on the left and appear in expandable `<details>` nodes on the right:

Cache

Search...

ROOT\_QUERY

Chapter:-1

Chapter:-2

Chapter:-3

Chapter:0

Chapter:1

Chapter:5

ROOT\_MUTATION

Review:5abe941d25aca9

Review:5abe960fc4eae1

Section:1-2

Section:1-intro

Section:5-1

Section:5-2

Section:foreword

Section:intro

GraphQL

Queries

Mutations

Cache

▼ ROOT\_QUERY

chapters: [Chapter]

0: ▶ Chapter:-3

1: ▶ Chapter:-2

2: ▶ Chapter:-1

3: ▶ Chapter:0

4: ▶ Chapter:1

5: ▶ Chapter:5

currentUser: User

email: "lorensr@gmail.com"

favoriteReviews:

firstName: "Loren"

hasPurchased: "FULL\_COURSE"

name: "Loren Sands-Ramshaw"

photo: "https://avatars2.githubusercontent.com/u/251288?v=4"

username: "lorensr"

githubStars: 10

reviews({"limit":10,"orderBy":"createdAt\_DESC"}): [Review]

0: ▶ Review:5abe960fc4eae1007fb7bc48

1: ▶ Review:5abe941d25aca9fe2306cff9

section({"id":"intro"}): Section

▶ Section:intro

## Chapter 6: React

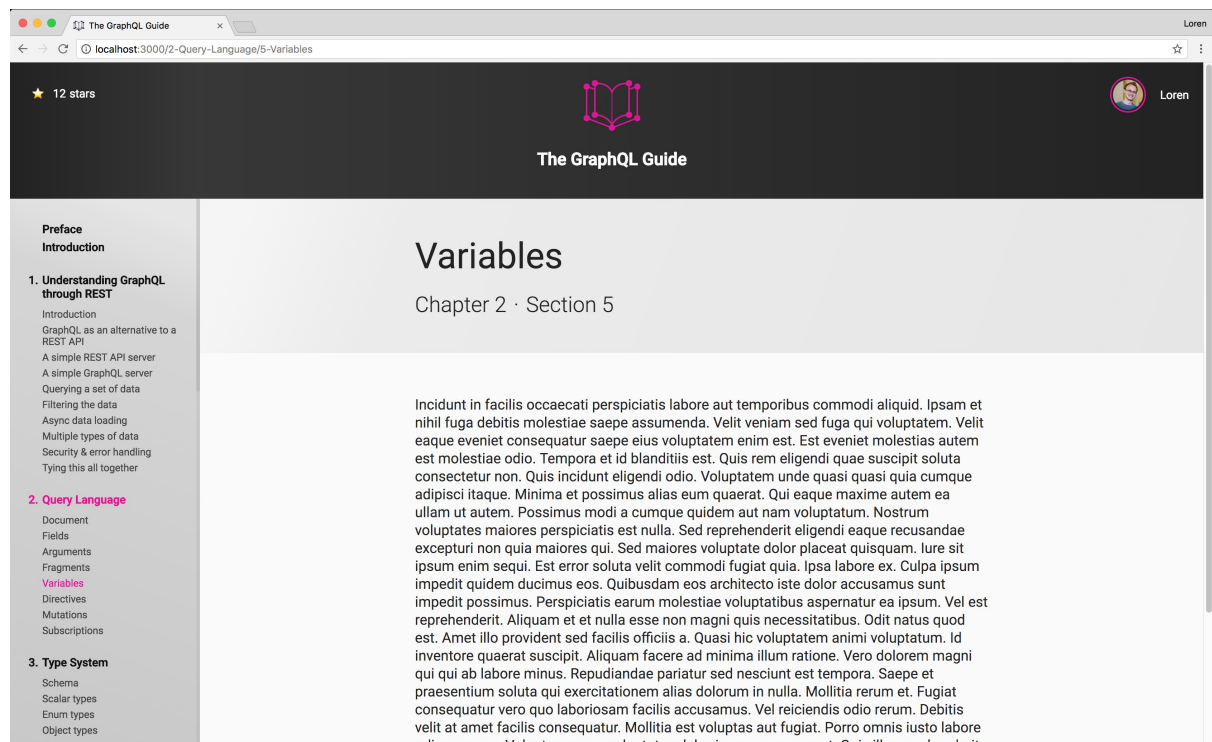
Chapter contents:

- [Setting up](#)
  - [Build options](#)
  - [App structure](#)
  - [Set up Apollo](#)
- [Querying](#)
  - [First query](#)
  - [Loading](#)
  - [Polling](#)
  - [Subscriptions](#)
  - [Lists](#)
  - [Query variables](#)
  - [Skipping queries](#)
- [Authentication](#)
  - [Logging in](#)
  - [Resetting](#)
- [Mutating](#)
  - [First mutation](#)
  - [Listing reviews](#)
  - [Optimistic updates](#)
  - [Arbitrary updates](#)
  - [Creating reviews](#)
  - [Using fragments](#)
  - [Deleting](#)
  - [Error handling](#)
  - [Editing reviews](#)
- [Advanced querying](#)
  - [Paginating](#)
    - [Offset-based](#)
      - [page](#)
      - [skip & limit](#)
    - [Cursors](#)
      - [after](#)
      - [orderBy](#)
  - [Updating multiple queries](#)
  - [Local state](#)
    - [Direct writes](#)
    - [Local mutations](#)
  - [REST](#)
  - [Review subscriptions](#)
    - [Subscription component](#)
    - [Add new reviews](#)
    - [Update on edit and delete](#)
  - [Prefetching](#)
    - [On mouseover](#)
    - [Cache redirects](#)

- [Batching](#)
- [Persisting](#)
- [Multiple endpoints](#)
- [Extended topics](#)
  - [Linting](#)
    - [Setting up linting](#)
    - [Fixing linting errors](#)
    - [Using linting](#)
  - [Uploading files](#)
  - [Testing](#)

Background: [single-page application](#), [HTTP](#), [Node](#), [git](#), [JSON](#), [JavaScript](#), [React](#)

In this chapter, we'll learn to use the [react-apollo](#) library through building the Guide web app—the code behind the <https://graphql.guide> site, where we can sign in, read the book, and write reviews. *[Beta note: the site isn't yet complete, so you'll see lorem ipsum in place of book content 😊.]* We'll go through setup, simple queries, complex queries, auth, and mutations for creating, updating, and deleting. Then we'll cover advanced topics like infinite scrolling, local state, SSR, working offline, and performance. Here's what it will look like:



## Setting up

Section contents:

- [Build options](#)
- [App structure](#)
- [Set up Apollo](#)

## Build options



Background: [server-side rendering](#)

In the early days, setting up a new React app was plagued by complex Webpack and Babel configurations. There are now a number of tools for this, four of which we recommend: Create React App, Gatsby, Next.js, and Meteor.

[Babel](#) converts our modern JavaScript to old JavaScript so it will run in the browser or Node. [Webpack](#) bundles our JavaScript and other files together into a website.

## Create React App

```
npm i -g create-react-app
create-react-app guide
cd guide/
npm start
```

[Create React App](#) (CRA) is a tool that configures Webpack and Babel to good, common defaults. For deployment, running `npm run build` gives us an `index.html`, our JavaScript bundle (or multiple bundles if we're code splitting), and imported static assets like CSS, images, and fonts.

## Gatsby

```
npm install -g gatsby-cli
gatsby new guide
cd guide/
gatsby develop
```

[Gatsby](#) is the best static site generator out there. But by “static site generator,” we don't mean it generates HTML-only noninteractive sites. It generates pages that render the HTML & CSS UI immediately and run JavaScript to hydrate the page into a React app. It can't generate logged-in content (like you can with SSR and cookies) because it's not your production server—deploying is building HTML, JS, and CSS files and serving them as-is (*statically*). However, you can [render logged-in content on the client](#).

## Next.js

```
npm i -g create-next-app
create-next-app guide
cd guide/
npm run dev
```

[Next.js](#) is similar to CRA in that it takes care of Webpack/Babel for us, but it also does [server-side rendering](#) (SSR), routing, automatic page-level code splitting, dynamic importing, and hot code reloading. CRA and Gatsby are just your dev server and build tool, whereas Next, since it does SSR, is also your Node production server.

Next does have an `export` command that outputs HTML and JS that you can serve as static files (like Gatsby), but the HTML is rendered once at the time that you run the `export` command, instead of in real time whenever a client requests the site.

## Meteor

<https://github.com/GraphQLGuide/guide.git>

```
curl https://install.meteor.com/ | sh
git clone https://github.com/jamiter/meteor-starter-kit.git guide
cd guide/
npm install
meteor
```

**Meteor** is similar to Next in that it is not only the build tool but also the production server. Unlike the other options, it does not use Webpack—it has its own advanced build system that is blissfully configuration-free. It does not have built-in SSR like Next does, but it does have dynamic imports, and all dynamically imported modules are fetched quickly over a WebSocket and [cached on the client](#) (in [IndexedDB](#)). It also does [differential bundling](#), further reducing bundle size for modern browsers.

## App structure

For our Guide app, we'll use CRA, because it's the most widely used and the most basic, straightforward option. Here's our starter app:

```
git clone https://github.com/GraphQLGuide/guide.git
cd guide/
git checkout 0_0.2.0
npm install
```

Now we should be able to run CRA's development server:

```
npm start
```

And see our app at [localhost:3000](http://localhost:3000):



To get started, edit `src/App.js`, and save to reload.

Our file structure is very similar to what we get when we run `create-react-app` :

```
.
├── .eslintrc
├── .gitignore
├── package-lock.json
├── package.json
├── public
└── favicon.ico
```

```

├── index.html
├── manifest.json
└── src
    ├── App.test.js
    ├── components
    │   └── App.js
    ├── index.css
    ├── index.js
    ├── jsconfig.json
    ├── logo.svg
    └── registerServiceWorker.js

```

`.eslintrc.js` — The CRA dev server ( `npm start` ) outputs linter warnings ([background on ESLint](#)), but it's nice to see the warnings directly in our text editor, so we have an `.eslintrc` file that uses the same rules as the dev server. Most editors' eslint plugins will pick this up, including [eslint](#) for our recommended editor, [VS Code](#).

`package.json`

```

{
  "name": "guide",
  "version": "0.2.0",
  "private": true,
  "dependencies": {
    "react": "^16.0.0",
    "react-dom": "^16.0.0",
    "react-scripts": "1.0.14",
    ...
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject",
    ...
  }
}

```

We have our normal react dependencies, `react` and `react-dom`, plus `react-scripts`, which is what CRA lives inside, and which provides the commands:

- `npm start` starts the dev server
- `npm run build` bundles app for deployment
- `npm test` runs all the tests found in `*.test.js` files
- `npm run eject` takes us out of CRA (replaces `react-scripts` in our `devDependencies` with a long list of other packages, adds a `scripts/` directory, and adds an 8-file `config/` directory with Webpack, Babel, and testing configuration)

In our `public/` directory, we have a [favicon](#), `manifest.json` (which is used when our app is added to an Android homescreen), and our only HTML page, `public/index.html` — our [SPA](#) shell, basically just:

```

<!doctype html>
<html lang="en">
  <head>
    <title>The GraphQL Guide</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>

```

We can add HTML, like meta tags to the head or the Google Analytics tracking script to the bottom of the body. Our React JavaScript code gets added to the body, and when it runs, it puts the app inside the root tag `<div id="root">`  
`</div>` :

src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'
import registerServiceWorker from './registerServiceWorker'

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

registerServiceWorker()

module.hot.accept()
```

Let's look at some of the lines:

`import './index.css'` — CRA supports importing CSS from JavaScript. There are many ways to do CSS with React, and we'll be sticking with this single plain `.css` file so that we can focus on the GraphQL parts of app-building.

`ReactDOM.render(<App />, document.getElementById('root'))` — Our only component, `<App />`, gets rendered into the `#root` div.

`registerServiceWorker()` — CRA includes a service worker (set up by `src/registerServiceWorker.js`) that caches our assets in the browser so that our app loads faster ([more info](#)).

`module.hot.accept()` — This enables HMR ([Hot Module Replacement](#)), a Webpack feature that updates JavaScript when code is saved in development without reloading the page.

Here's our App component:

src/components/App.js

```
import React, { Component } from 'react'
import logo from './logo.svg'

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">The GraphQL Guide</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code>, and save to reload.
        </p>
      </div>
    )
  }
}

export default App
```

`import logo from './logo.svg'` — CRA supports importing files, like images and fonts. When we import a file, it gets included in the app bundle, and we get a URL that we can use—for example, in a `src` attribute:

```
<img src={logo} className="App-logo" alt="logo" />
```

We also have a test file:

src/App.test.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
```

```
it('renders without crashing', () => {
  const div = document.createElement('div')
  ReactDOM.render(<App />, div)
})
```

This and any other files ending in `.test.js` get run when we do `npm test`.

The last thing in `src/` is our `jsconfig.json` file, which tells VS Code what type of JavaScript we're using (CRA includes, for example, `async/await` from ES2017) and where it's located.

## Set up Apollo

The best GraphQL library for React is `react-apollo`. It has all the features we talked about in the [Client Libraries](#) section and more. Our `package.json` already has these packages, but normally we would install `react-apollo` and its associated packages with:

```
npm i -S react-apollo graphql graphql-tag apollo-client apollo-cache-inmemory apollo-link-http
```

Now we need to create an instance of `ApolloClient` and wrap our app JSX in a component called `<ApolloProvider>`, which provides our client instance to all descendants. So we go to `src/index.js`, where our `<App />` component is rendered, and replace the `ReactDOM.render` line:

`src/index.js`

```
import { ApolloClient } from 'apollo-client'
import { ApolloProvider } from 'react-apollo'
import { InMemoryCache } from 'apollo-cache-inmemory'
import { createHttpLink } from 'apollo-link-http'

const link = createHttpLink({
  uri: 'https://api.graphql.guide/graphql'
})

const cache = new InMemoryCache()

const client = new ApolloClient({ link, cache })

ReactDOM.render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  document.getElementById('root')
)
```

We highly recommend typing out the code instead of copy/pasting—you'll learn it better! 🤔👉👏

We tell `ApolloClient` where to send queries by giving it a network link pointed at our GraphQL server—in this case `https://api.graphql.guide/graphql`.

## Querying

Section contents:

- [First query](#)
- [Loading](#)
- [Polling](#)
- [Subscriptions](#)

- [Lists](#)
- [Query variables](#)
- [Skipping queries](#)

## First query

One of the fields we can query for is `githubStars`, the number of stars the Guide's [github repo](#) has. Let's look at how we can make that query and display the results. We'll start out by adding a component to display the star count:

`src/components/StarCount.js`

```
import React from 'react'

const StarCount = ({ githubStars }) => {
  return (
    <a className="StarCount" href="https://github.com/GraphQLGuide/guide">
      {githubStars}
    </a>
  )
}
```

But how do we get `githubStars` as a prop? First we write the query, which is pretty simple, since it's a top-level [scalar](#) query field:

```
import gql from 'graphql-tag'

const STARS_QUERY = gql`
  query StarsQuery {
    githubStars
  }
`
```

We name it `STARS_QUERY` because convention is to use all caps for query constants. We use an [operation name](#) ( `StarsQuery` ) so that it's easier to find and debug. `gql` is a [template literal tag](#) that parses our [query document](#) string, converting it into a structured object that we can pass to Apollo—now we can give it to Apollo's `<Query>` component:

```
import PropTypes from 'prop-types'
import { Query } from 'react-apollo'

const StarCount = ...

StarCount.propTypes = {
  githubStars: PropTypes.number,
  loading: PropTypes.bool.isRequired
}

export default () => (
  <Query query={STARS_QUERY}>
    {({ data: { githubStars }, loading }) => (
      <StarCount githubStars={githubStars} loading={loading} />
    )}
  </Query>
)
```

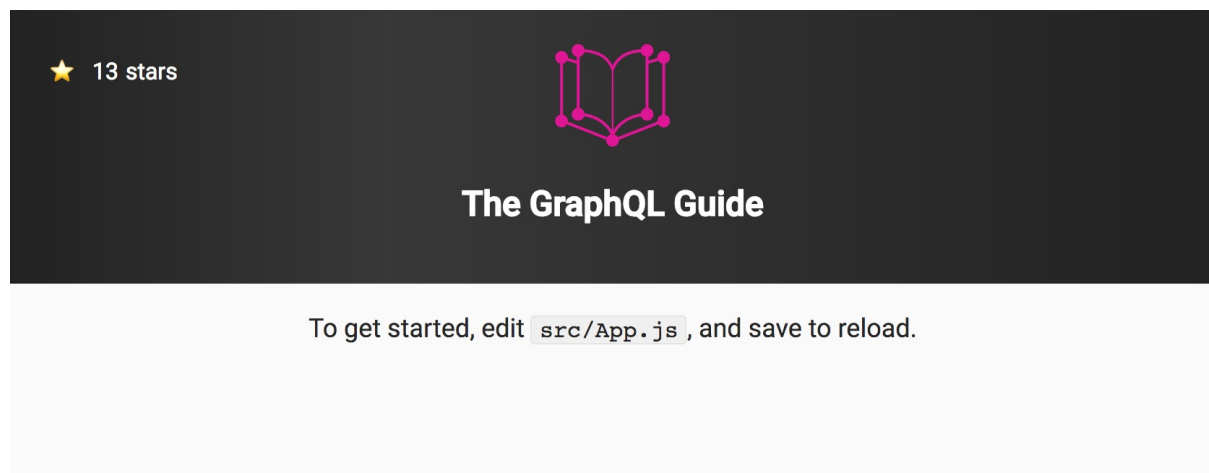
It follows the [render prop](#) pattern, in which `Query` calls `children` as a function, giving the function an object argument with information about the query. There are [many object attributes](#) we can choose from, but for now, we'll just use `data` (the `"data"` attribute in the response from the server) and `loading`, a boolean that lets us know when we're waiting for the response to arrive from the server. In our `children` function (which is called the render prop), we pass those two arguments as props to `<StarCount>`.

When the page is loaded and the `<Query>` component is created, Apollo will send the query to the server and give the result to our render prop. Now we can add the component to our app:

`src/components/App.js`

```
import StarCount from './StarCount'
...
<header className="App-header">
  <StarCount />
  <img src={logo} className="App-logo" alt="logo" />
  <h1 className="App-title">The GraphQL Guide</h1>
</header>
```

And we have a working GraphQL-backed app!



`react-apollo` provides two APIs for making queries—the `<Query>` render prop API, and the HOC (higher-order component) API. Which we use is mostly a matter of preference—the one thing the render prop API can do that the HOC API can't is use a dynamic query. The aesthetic differences are whether we're providing querying options and child component props in JSX (render prop) or JS objects (HOC), and whether we combine multiple queries by nesting JSX or composing HOCs. Since we'll be writing components that use multiple queries and mutations, and we have limited horizontal width in our ebook readers, we'll avoid highly nested JSX by mostly using HOCs. Note that it's easy to translate between the two APIs, since they have the same props and provide us with the same information. So when we learn one API, we learn both.

Here's the same component using `react-apollo`'s `graphql()` HOC API:

```
import { graphql } from 'react-apollo'

const StarCount = ...

const withData = graphql(STARS_QUERY, {
  props: ({ data: { githubStars, loading } }) => ({
    githubStars,
    loading
  })
})

export default withData(StarCount)
```

`graphql()` creates an HOC, which we call `withData` and use to wrap `StarCount`. The first parameter to `graphql()` is the query (which is why with the HOC API, we can't change the query), and the second is an optional config object. Here we're using the `props` config function, which tells Apollo which props we want our component to be given. The function gets the query response, in the form:

```
{
  data: {
    githubStars: 1,
    loading: false,
    other things...
  }
}
```

and returns which props we want—in this case `githubStars` and `loading`—instead of the default `data` prop with a long list of things.

## Loading

If you're jumping in here, `git checkout 1_0.2.0` (tag `1_0.2.0`, or compare `1...2`)

When we reload the app, we see a flash of “:star: stars” before the number appears, pushing `stars` to the right. When `<StarCount>` is rendered the first time, it doesn't have the number of stars yet, but it tells Apollo to send the query. Once the query response comes back from the server, it is re-rendered—this time, with the number. Let's log to see it happening:

`src/components/StarCount.js`

```
const StarCount = ({ githubStars, loading }) => {
  console.log(
    'rendering StarCount',
    `githubStars: ${githubStars}, loading: ${loading}`
  )
}
```

```
rendering StarCount
githubStars: undefined, loading: true
rendering StarCount
githubStars: 8, loading: false
```

We see that it's rendered twice—first `loading` is `true` and `githubStars` is `undefined`, and then later, once the query has finished, `loading` is `false` and `githubStars` has a value.

:star: stars without a number doesn't make sense, and `stars` jumping to the right when the number appears doesn't look nice, so let's hide everything until the number has arrived by adding the modifier CSS class `'loading'` when the `loading` prop is `true`:

```
import classNames from 'classnames'

const StarCount = ({ githubStars, loading }) => {
  return (
    <a className={classNames('StarCount', { loading })}>
```

`classNames` takes strings or objects as arguments and combines them into a React `className` string. For objects, it includes the key if the value is true. For example, `classNames('a', { b: false, c: true }, 'd')` returns `'a c d'`.

When `loading` becomes `false`, the CSS class `'loading'` is removed, and the component fades in.

## Polling



Right now our star count is static—once it's fetched, the number remains on the page until the page is refreshed. If the actual number of stars on the repository changes, we won't know until we refresh. If we want to keep the number (and any other GraphQL data) up to date, we can do so in two different ways: polling and [subscriptions](#). Polling is much easier to implement, so let's do that first. We can add a `pollInterval` prop to our query in `StarCount.js`:

`src/components/StarCount.js`

```
<Query query={STARS_QUERY} pollInterval={5 * 1000}>
  {{{ data: { githubStars }, loading }}} => (
    <StarCount githubStars={githubStars} loading={loading} />
  )
</Query>
```

Now every five seconds, Apollo will resend our `STARS_QUERY`. If the response has a different value for `githubStars`, Apollo will pass us the new prop, which will trigger a component re-render, and the new count will be displayed on the page.

The equivalent `pollInterval` option with `graphql()` is:

```
const withData = graphql(STARS_QUERY, {
  options: { pollInterval: 5 * 1000 },
  props: ...
})
```

Depending on what type of data we're keeping up to date, we may want to use some kind of visual cue or animation when it changes. There are a few possible motivations for this:

1. Calling attention to the change to make the user aware that it happened—a common example in this category is the brief yellow background glow. Another example is in Google Docs—the colored cursor labeled with a name that follows someone's live edits. However, sometimes a user doesn't need to know that a piece of data has changed, and calling attention to it would needlessly distract them from what they were paying attention to.
2. Making the change visually smoother. If a change in the data triggers some node on the page to change in size, and there are other nodes on the page around it, the other nodes might jump to a new location when the browser reflows—for example, if the data is a paragraph of text, and the updated paragraph is twice as long, everything below that paragraph will be pushed down. We can make this change look nicer by animating the data container to its new size and animating the displaced components to their new locations. This also gives time for the user to notice which part of the page changed, which is helpful for situations in which the user doesn't realize why things on the page jumped around.
3. For fun 😊. Animations can be fun, and sometimes we add them just because we like how it feels.

The data change that happens in our app is a number that is usually just going up by 1. This type of change is well-suited to an odometer animation, where each digit is on a number wheel that rotates up or down to reveal the next number. The benefit of this animation is #3, and the downside is #1—the odometer changing draws more attention to the change than a non-animated change does, but the user doesn't need to know when the star count changes (they're just trying to read the book!). So we might not add this animation to a serious app, but let's add it to our app for fun 😊. It's easy with the `react-odometerjs` component:

`src/components/StarCount.js`

```
import Odometer from 'react-odometerjs'

...

<a
  className={classNames('StarCount', { loading })}
  href="https://github.com/GraphQLGuide/guide"
  target="_blank"
  rel="noopener noreferrer"
>
```

```
{githubStars && <Odometer value={githubStars} />}
</a>
```

Now when the polling `STARS_QUERY` results in a new `githubStars` value, we pass the new number to the `<Odometer>` component, which does the animation.

We need the truth guard (`githubStars &&`) because `<Odometer>` throws an error when it's given an `undefined` value (and as we found out [before](#) when logging, `githubStars` starts out `undefined`).

We can test it out by starring and un-starring the [repository on GitHub](#) and watching the number in our app update.

## Subscriptions

Background: [webhooks](#)

If you're jumping in here, `git checkout 2_0.2.0` (tag `2_0.2.0`, or compare `2...3`)

When we poll for new data every 5 seconds, it takes 2.5 seconds on average (as little as 0, and as much as 5) for a change to show up, plus a little time for the server to talk to GitHub and get the response back to us. For certain types of apps, like a chat app or multiplayer games, it's important to receive updates in less than 2.5 seconds. One thing we can do is reduce the poll interval—for instance, a 500 ms interval would mean an average update speed of 250 ms (plus server response time). This would be fast enough for a chat app but not fast enough for some games. And it comes at a certain cost in server workload (it now has to respond to 10 times as many requests) and browser workload (sending requests takes up main-thread JavaScript time, perhaps during one of the [10ms windows](#) in which the thread needs to quickly calculate a 60 fps animation). So while polling is often the best choice given its simplicity to implement (we just added that single `pollInterval` option), sometimes we want something more efficient and real-time.

In these cases we can use GraphQL [subscriptions](#), in which our server will send us updates to our data as they occur. The main drawback to subscriptions is that it takes extra work to implement on the server. (In the next chapter we'll learn how to [add subscription support](#).) Another possible drawback is that if the subscription data changes frequently, it can hurt client performance by taking up time receiving, updating the store, and re-rendering the page.

While GraphQL servers can support different methods of transporting subscription updates to clients (the GraphQL spec is transport-agnostic), the usual method is over WebSockets.

*WebSocket* is a format for sending messages over the internet (like [HTTP](#)). It allows for very fast two-way communication by keeping a connection open and allowing the server to initiate messages to the client.

We could replace our HTTP link with a WebSocket link in `index.js` using the `apollo-link-ws` package:

```
import { WebSocketLink } from 'apollo-link-ws'

const link = new WebSocketLink({
  uri: `ws://localhost:5000/`,
  options: {
    reconnect: true
  }
})
```

This would establish a WebSocket connection that remains open for the duration of the client session, and all GraphQL communication (queries, mutations, and subscriptions) would be sent over the connection. However, authentication over the WebSocket is a little involved, so we'll go with a hybrid transport solution: we'll send queries and mutations over an HTTP link (which we'll add auth to later), and we'll send subscriptions over the unauthenticated WebSocket link. We can do this because all of the data used in the Guide's real-time features (for example `starCount`, and later on, `reviews`) is public.

src/index.js

```
import { split } from 'apollo-link'
import { WebSocketLink } from 'apollo-link-ws'
import { getMainDefinition } from 'apollo-utilities'

const httpLink = createHttpLink({
  uri: 'https://api.graphql.guide/graphql'
})

const wsLink = new WebSocketLink({
  uri: `wss://api.graphql.guide/subscriptions`,
  options: {
    reconnect: true
  }
})

const link = split(
  ({ query }) => {
    const { kind, operation } = getMainDefinition(query)
    return kind === 'OperationDefinition' && operation === 'subscription'
  },
  wsLink,
  httpLink
)
```

The `ApolloClient` constructor options object takes a single link, so we need to compose our two links together. We can use the `split()` function, which takes a function and two links. The function is given the current query, and if it returns true, the first link is used for the query; otherwise, the second is used. In our `split()` function we look up the query operation and return true if it's a subscription query, which directs the query to the WebSocket link `wsLink`.

Now we can subscribe to updates to the star count with this simple subscription:

src/components/StarCounts.js

```
const STARS_SUBSCRIPTION = gql`
  subscription StarsSubscription {
    githubStars
  }
`
```

To start the subscription, we use a function `subscribeToMore` that `react-apollo` provides us:

src/components/StarCounts.js

```
StarCount.propTypes = {
  githubStars: PropTypes.number,
  loading: PropTypes.bool.isRequired,
  subscribeToMore: PropTypes.func.isRequired
}

...

<Query query={STARS_QUERY} pollInterval={5 * 1000}>
  ({ data: { githubStars }, loading, subscribeToMore }) => (
    <StarCount
      githubStars={githubStars}
      loading={loading}
      subscribeToMore={subscribeToMore}
    />
  )
</Query>
```

And then we can use it in our component. We want to start the subscription when the component is initialized (in `componentDidMount`), so we need to convert `StarCounts` from a functional component to a class that can have lifecycle methods:

src/components/StarCounts.js

```
class StarCount extends React.Component {
  componentDidMount() {
    this.props.subscribeToMore({
      document: STARS_SUBSCRIPTION,
      updateQuery: (
        previousResult,
        { subscriptionData: { data: { githubStars } } }
      ) => ({ githubStars })
    })
  }

  render() {
    const { githubStars, loading } = this.props
    ...
  }
}
```

`subscribeToMore` takes the GraphQL document specifying our subscription and an `updateQuery` function.

`updateQuery` is called each time the client receives new subscription data from the server. It's given the result of the previous query ( `STARS_QUERY` in our case) and the subscription data, and it returns an updated query result, which is used to provide new props to the component. In our case, we're just replacing the old result with the GitHub star count received in the `subscriptionData`. But if GitHub never lets us un-star repos, and the star count only ever increased, then we might use a `justGotStarred` subscription that published `{ newStar: true }` to the client. Then our `updateQuery` would look like:

```
this.props.subscribeToMore({
  document: JUST_GOT_STARRED_SUBSCRIPTION,
  updateQuery: (previousResult, update) {
    return {
      githubStars: previousResult.githubStars + 1
    }
  }
})
```

The last thing we need to do is test whether our `STARS_SUBSCRIPTION` is working: we stop polling by removing the `pollInterval` prop from our `<Query>` in `StarCount.js`:

```
<Query query={STARS_QUERY} pollInterval={5 * 1000}>
```

Now we can star and unstar the [Guide repo](#) and see the count quickly change in our app. We might notice a slight delay sometimes, and that's because the server is polling the GitHub API once a second for updates, so the subscription data reaching the client could be as old as 1 second plus network time. We could improve this by reducing the polling interval on the server or by setting up a [webhook](#)—the most efficient and lowest-latency solution, in which the only delay would be network time: GitHub would immediately notify our server of the change, and the server would immediately send the subscription update over the WebSocket to the client.

The equivalent query using `graphql()` is:

```
const withData = graphql(STARS_QUERY, {
  props: ({ data: { githubStars, loading, subscribeToMore } }) => ({
    githubStars,
    loading,
    subscribeToMore
  })
})
```

## Lists

If you're jumping in here, `git checkout 3_0.2.0` (tag `3_0.2.0`, or compare `3...4`)

See the [Listing reviews](#) section for another example of querying a list of data.

Next let's get to the heart of our app—the stuff below the header! We'll want to reserve most of the space for the book content, since there's a lot of it, and reading it is the purpose of the app 😊. But let's put a thin sidebar on the left for the table of contents so that readers can easily navigate between sections.

To begin, we replace the `<p>` in `<App>` with the two new sections of the page:

`src/components/App.js`

```
import TableOfContents from './TableOfContents'
import Section from './Section'

...

<div className="App">
  <header className="App-header">
    <StarCount />
    <img src={logo} className="App-logo" alt="logo" />
    <h1 className="App-title">The GraphQL Guide</h1>
  </header>
  <TableOfContents />
  <Section />
</div>
```

We call the second component `Section` because it will display a single section of a chapter at a time. Let's think about the loading state first—we'll be fetching the table of contents and the section content from the API. We could do a loading spinner, but a nicer alternative when we're waiting for text to load is a loading skeleton—an animated gray bar placed where the text will appear. Let's put a few bars in both components:

`src/components/Section.js`

```
import React from 'react'
import Skeleton from 'react-loading-skeleton'

const Section = ({ loading = true }) => (
  <section className="Section">
    <div className="Section-header-wrapper">
      <header className="Section-header">
        <h1>Title</h1>
        <h2>Subtitle</h2>
      </header>
    </div>
    <div className="Section-content">
      {loading ? <Skeleton count={7} /> : null}
    </div>
  </section>
)

export default Section
```

`<Section>` isn't being passed a `loading` prop yet, since we haven't used the `graphql()` function to attach a query to the component yet, but we can give `loading` a default value of `true` for now so that we can work with it. `count={7}` will give us 7 gray bars, representing 7 lines of text. Now for the sidebar:

`src/components/TableOfContents.js`

```
import React from 'react'
import Skeleton from 'react-loading-skeleton'

const TableOfContents = ({ loading = true }) => {
  return (
    <nav className="TableOfContents">
      {loading ? (
```

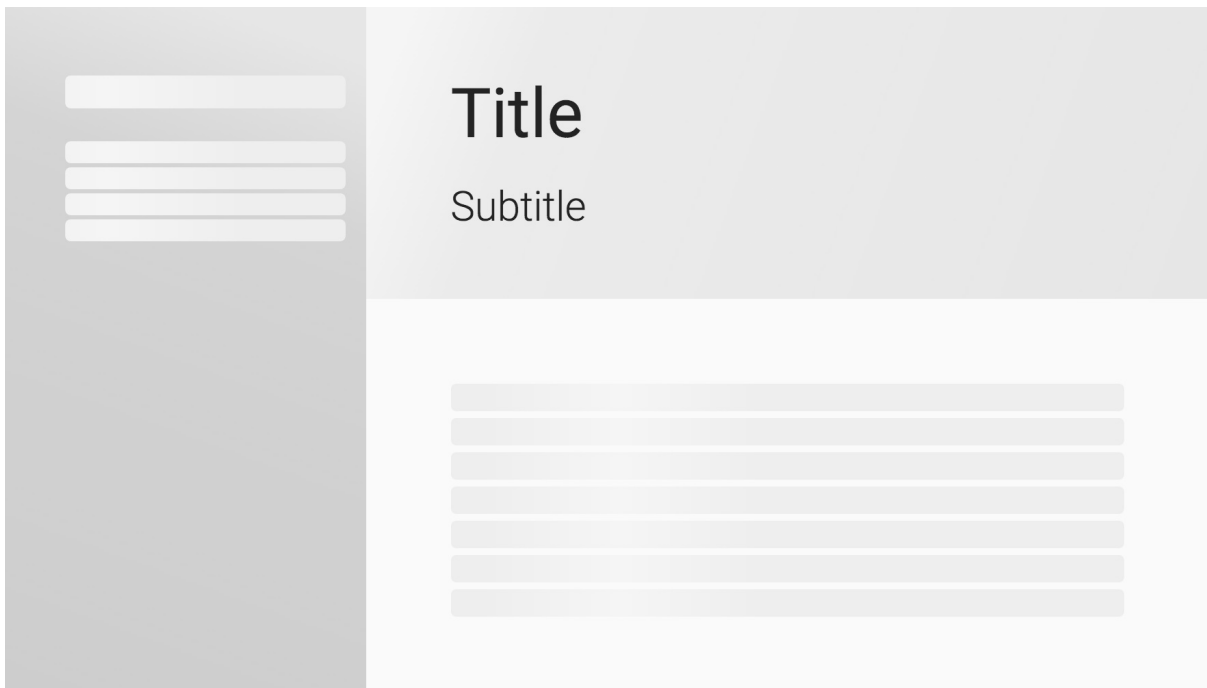
```

    <div>
      <h1>
        <Skeleton />
      </h1>
      <Skeleton count={4} />
    </div>
  ) : null}
</nav>
)
}

export default TableOfContents

```

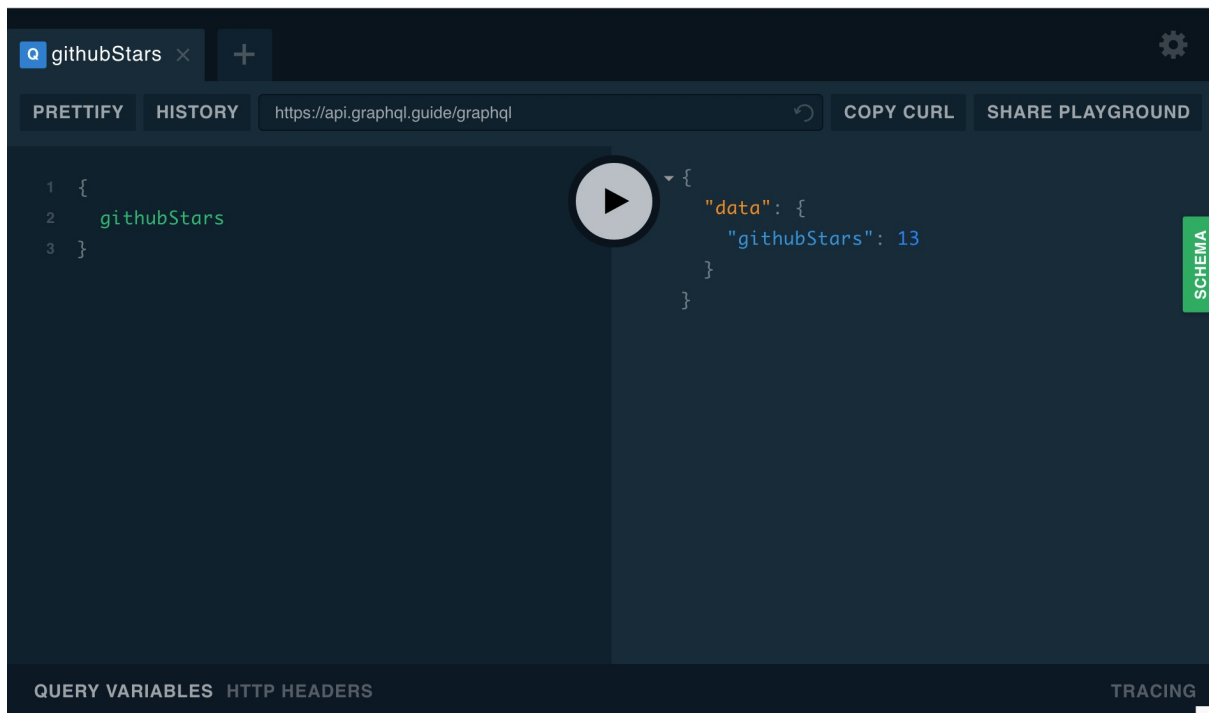
`<Skeleton>` picks up the surrounding font size, so we'll see a larger gray line (in place of a chapter title) and then 4 smaller lines (in place of section titles):



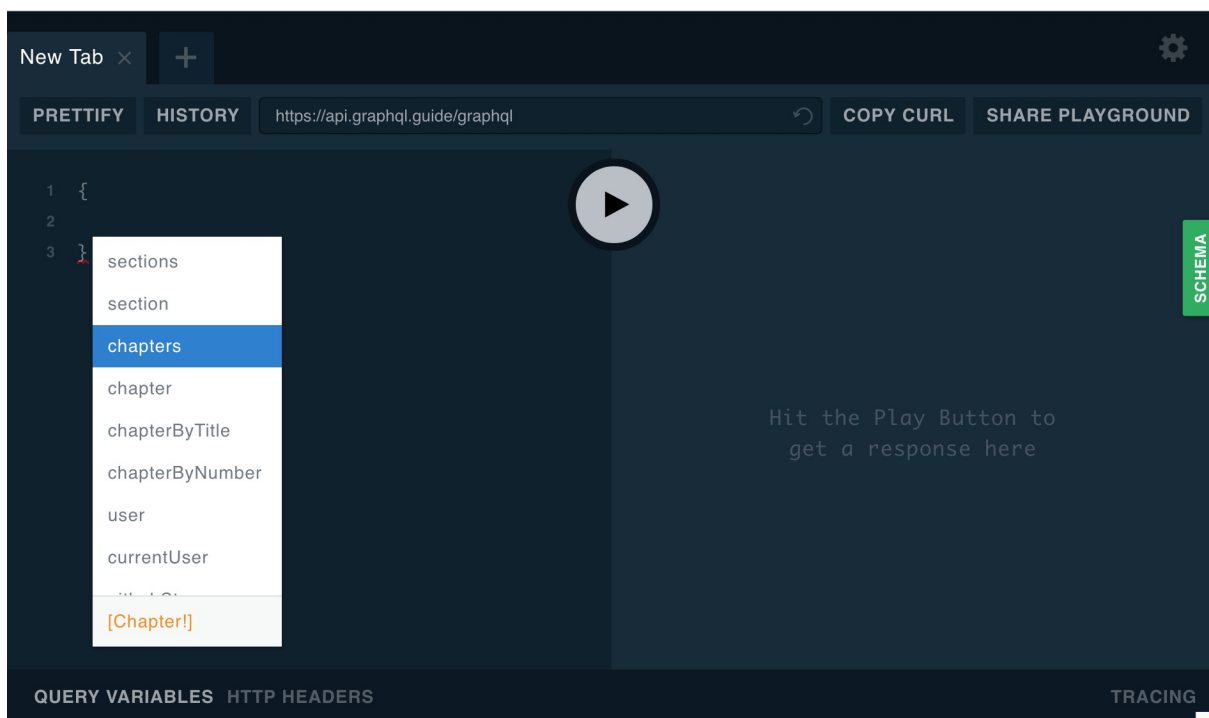
Now let's construct the query for the data we need to display in `TableOfContents`. We can explore the Guide API's schema in GraphQL Playground, an IDE for writing GraphQL queries. For instance, here we're querying for `{ githubStars }`:

Playground: `query { githubStars }`

On the left side we have the GraphQL document, and when we click the play button (or `command-return`), we see the response on the right:



Now let's delete `githubStars`, and with our cursor in between the `query` braces, we hit `control-space` to bring up query suggestions:



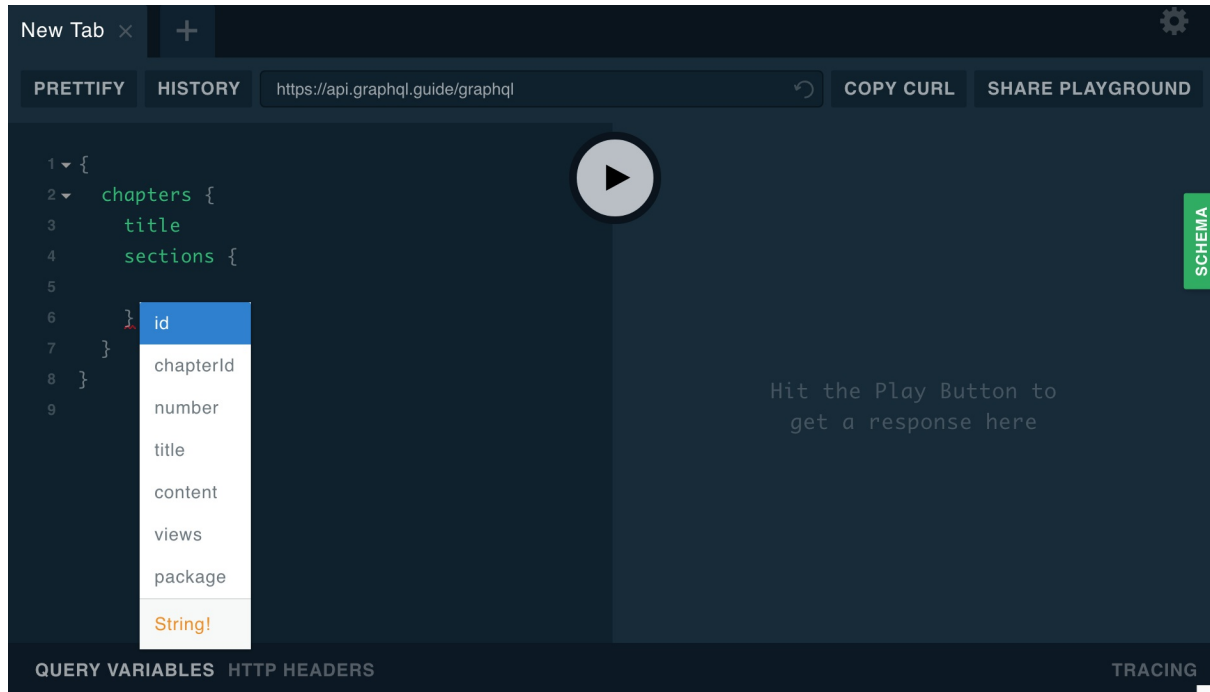
The one we want is `chapters`. Now we can add an inner set of braces (the `selection set` on `chapters`), move our cursor inside, and hit `control-space` again to see the available fields of a `Chapter` (which is the type that `chapters` returns):

```
query {
  chapters {
  }
}
```

We'll want to display the `title` and the `sections`, and we do the same to see which fields of a `Section` we want.

```
query {
  chapters {
    title
    sections {
    }
  }
}
```

And we see `title`, which we will want for each section.

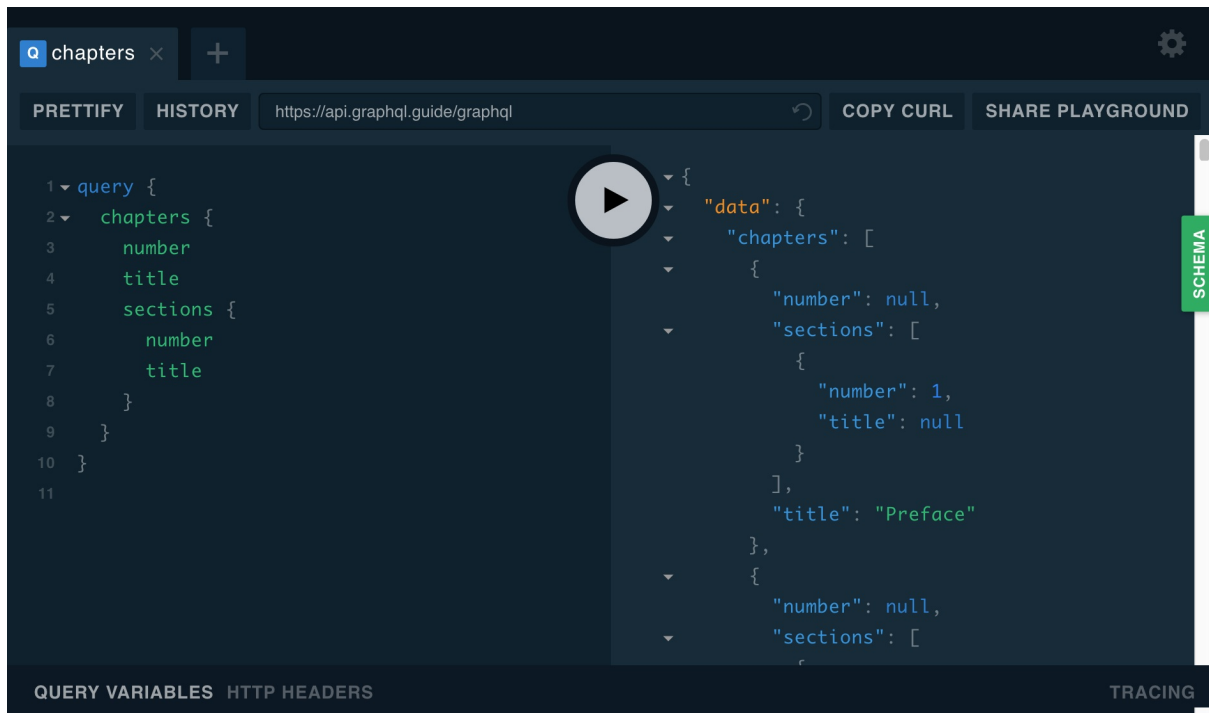


We will also want to display the chapter and section numbers, so let's add those as well. Our whole query is:

```
query {
  chapters {
    number
    title
    sections {
      number
      title
    }
  }
}
```

We can see what the data looks like by hitting the play button or `command-return`.





To attach the query to our component, we give it a name, `ChapterQuery`, put it inside a `gql`` template string, and use `graphql()` to specify what props our HOC will give to `TableOfContents`:

`src/components/TableOfContents.js`

```

import { graphql } from 'react-apollo'
import gql from 'graphql-tag'

const CHAPTER_QUERY = gql`
  query ChapterQuery {
    chapters {
      id
      number
      title
      sections {
        id
        number
        title
      }
    }
  }
`

const withData = graphql(CHAPTER_QUERY, {
  props: ({ data: { chapters, loading } }) => ({
    chapters,
    loading
  })
})

export default withData(TableOfContents)

```

And now that our component will be getting props, we add prop types:

`src/components/TableOfContents.js`

```

import PropTypes from 'prop-types'

TableOfContents.propTypes = {
  chapters: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,

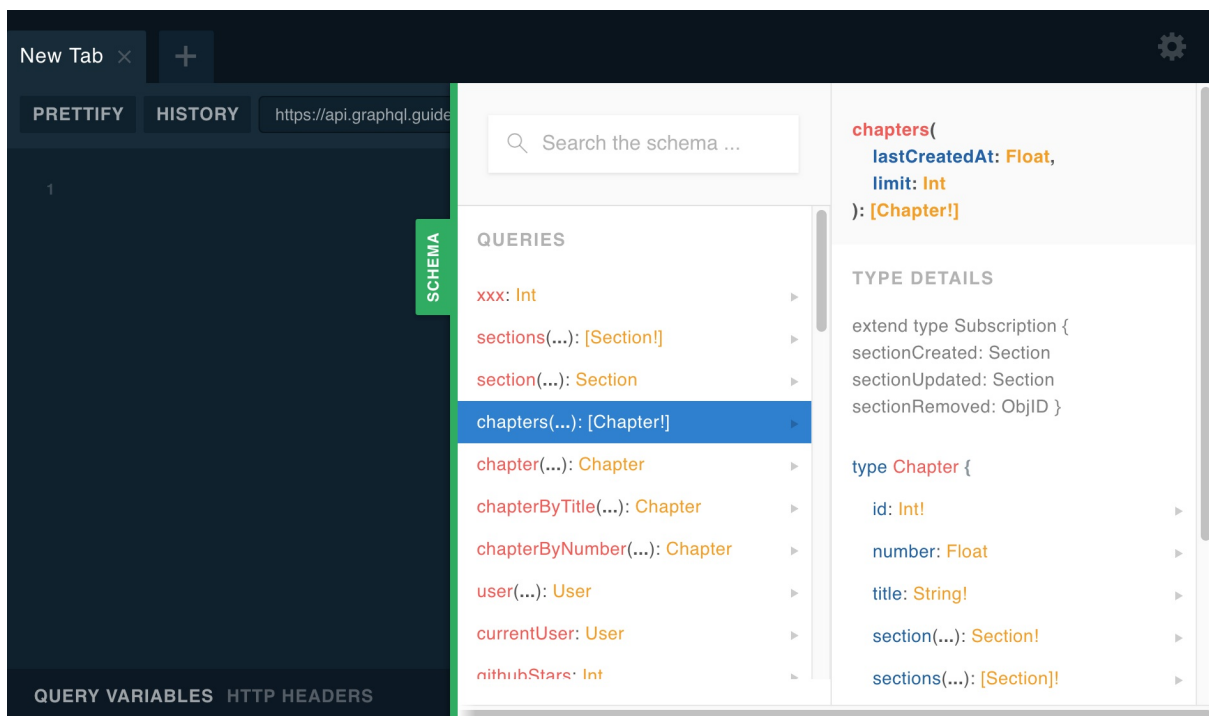
```

```

    number: PropTypes.number,
    title: PropTypes.string.isRequired,
    sections: PropTypes.arrayOf(
      PropTypes.shape({
        id: PropTypes.number.isRequired,
        number: PropTypes.number.isRequired,
        title: PropTypes.string
      }).isRequired
    ).isRequired
  }).isRequired
),
  loading: PropTypes.bool.isRequired
}

```

We know that we can add `.isRequired` to the fields of `chapters` and `sections` because we can see in the Guide schema that they're non-null—in Playground, we click "SCHEMA" on the right to open up the schema tab, click on `chapters`, and notice that, for example `title: String!` has an exclamation mark, so it will always have a value.



This means that when our component is provided `chapters`, the `title` field will always be present.

Playground: `query { chapters { number title sections { number title } } }`

Note that `chapters: PropTypes.arrayOf(...)` doesn't have an `.isRequired`, because initially, while `loading` is `true`, `chapters` is `undefined`.

Next let's use the new props our component gets. We can remove the default `true` value for `loading` and add `chapters`. For each chapter we display a list of links to each section:

`src/components/TableOfContents.js`

```

import { NavLink } from 'react-router-dom'
import classNames from 'classnames'

import { slugify, withHyphens } from '../lib/helpers'

const LoadingSkeleton = () => (
  <div>
    <h1>
      <Skeleton />
    </h1>
    <Skeleton count={4} />
  </div>
)

```

```

    </div>
  )

  const TableOfContents = ({ chapters, loading }) => (
    <nav className="TableOfContents">
      {loading ? (
        <LoadingSkeleton />
      ) : (
        <ul className="TableOfContents-chapters">
          {chapters.map(chapter => {
            const chapterIsNumbered = chapter.number !== null
            return (
              <li
                className={classNames({ numbered: chapterIsNumbered })}
                key={chapter.id}
              >
                <NavLink
                  to={{
                    pathname: slugify(chapter),
                    state: { chapter, section: chapter.sections[0] }
                  }}
                  className="TableOfContents-chapter-link"
                  activeClassName="active"
                  isActive={(match, location) => {
                    const rootPath = location.pathname.split('/')[1]
                    return rootPath.includes(withHyphens(chapter.title))
                  }}
                >
                  {chapterIsNumbered && (
                    <span className="TableOfContents-chapter-number">
                      {chapter.number}
                    </span>
                  )}
                  {chapter.title}
                </NavLink>
                {chapterIsNumbered && (
                  <ul className="TableOfContents-sections">
                    {chapter.sections.map(section => (
                      <li key={section.number}>
                        <NavLink
                          to={{
                            pathname: slugify(chapter, section),
                            state: { chapter, section }
                          }}
                          className="TableOfContents-section-link"
                          activeClassName="active"
                        >
                          {section.title}
                        </NavLink>
                      </li>
                    ))}
                  </ul>
                )}
              </li>
            )
          })}
        </ul>
      )}
    </nav>
  )

```

Okay, so that was a lot of code 😊. We've got an outer list of chapters, and for each chapter we have an inner list of sections. We've got React Router `<NavLink>` s that add an `"active"` class when the URL matches the link path. And we use the `slugify()` helper to generate paths.

[src/lib/helpers.js](#)

```

export const withHyphens = string => string.replace(/ /g, '-')

// generate paths of the form:
// `~/Forward`
// `~/Preface`

```

```
// `/1-Understanding-GraphQL-through-REST/1-Introduction`
export const slugify = (chapter, section) => {
  if (!section) {
    if (chapter.sections.length) {
      // default to the first section
      section = chapter.sections[0]
    } else {
      return '/' + withHyphens(chapter.title)
    }
  }

  const chapterSlug = chapter.number + '-' + withHyphens(chapter.title)
  const sectionSlug = section.number + '-' + withHyphens(section.title)
  return `/${chapterSlug}/${sectionSlug}`
}
```

Also, to get React Router working, we need to wrap our app in `<BrowserRouter>`:

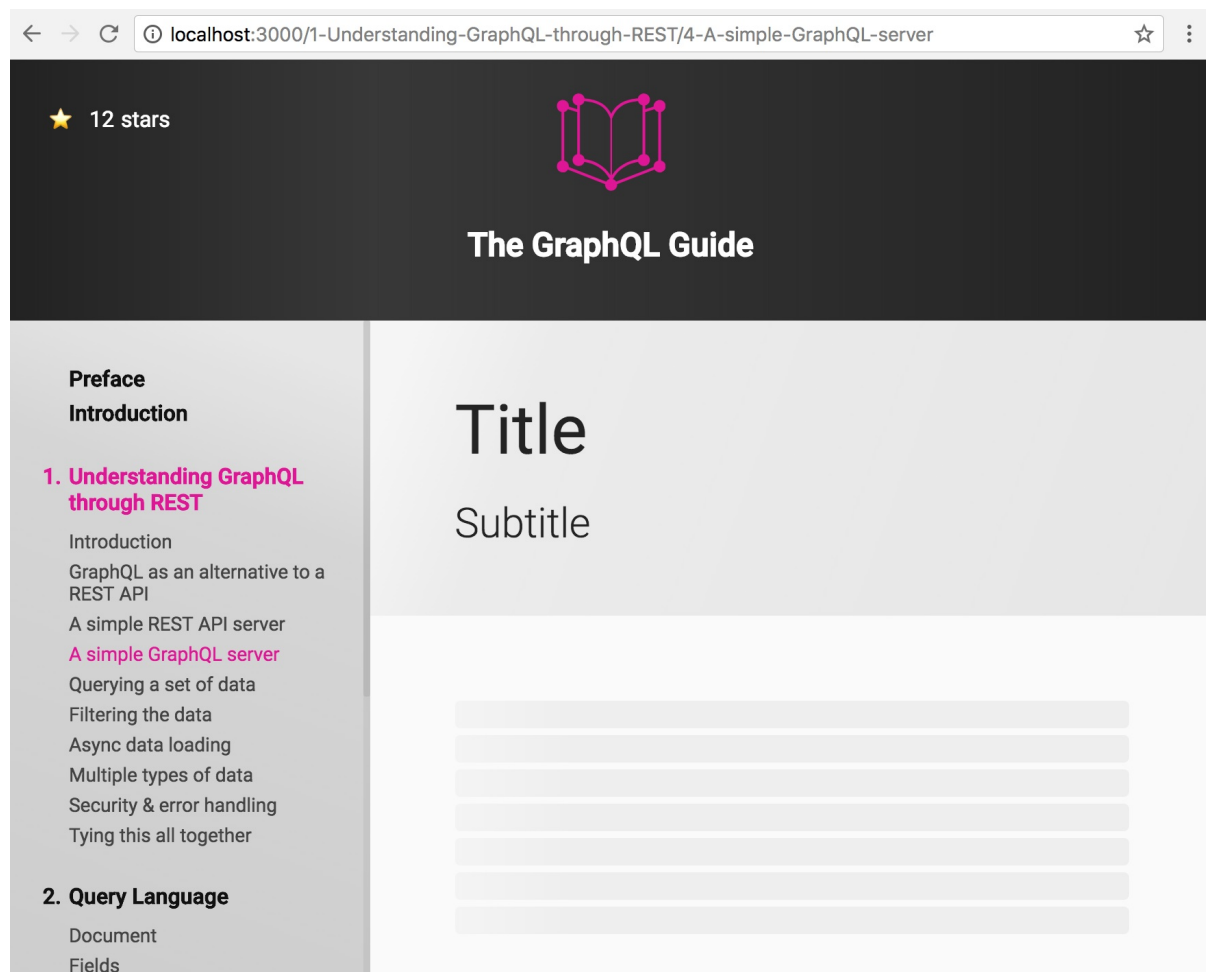
`src/index.js`

```
import { BrowserRouter } from 'react-router-dom'

ReactDOM.render(
  <BrowserRouter>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </BrowserRouter>,
  document.getElementById('root')
)
```

With all this JSX code, we're starting to feel the best thing about GraphQL on the client side—that most of the coding is in the view instead of in data fetching. We don't have a bunch of REST endpoint fetching and parsing and caching and passing code; instead, we attach simple query strings to the components that need them, and we get the data in the props.

Now we should see the table of contents on the left side of the page, and we can click between sections and see the active links and path changing:



## Query variables

If you're jumping in here, `git checkout 4_0.2.0` (tag `4_0.2.0`, or compare `4...5`)

Let's fill in the book content next! Say we have a section ID, like `'intro'` —how do we get the content? Let's look in Playground to find the right query to make:

Playground: `query { }`

There's a `section(id: String!)` query that returns a `Section` object, which has a `content` field. So let's try it out:

Playground: `query { section(id: "intro") { content } }`

Next we add the query to our component:

`src/components/Section.js`

```
import PropTypes from 'prop-types'
import { graphql } from 'react-apollo'
import gql from 'graphql-tag'

Section.propTypes = {
  section: PropTypes.shape({
    content: PropTypes.string.isRequired
  }),
  loading: PropTypes.bool.isRequired
}

const SECTION_QUERY = gql`
  query SectionContent {
```

```

    section(id: "intro") {
      content
    }
  }
}

const withData = graphql(SECTION_QUERY, {
  props: ({ data: { section, loading } }) => ({ section, loading })
})

export default withData(Section)

```

Now our component should get a `section` prop that will have the same `content` string that we saw returned in Playground, and we can use it.

`src/components/Section.js`

```

const Section = ({ loading, section }) => (
  <section className="Section">
    <div className="Section-header-wrapper">
      <header className="Section-header">
        <h1>Title</h1>
        <h2>Subtitle</h2>
      </header>
    </div>
    <div className="Section-content">
      {loading ? <Skeleton count={7} /> : section.content}
    </div>
  </section>
)

```

We can read the book! 📖 But we've got a hard-coded section ID—let's turn our `section(id)` argument into a variable:

`src/components/Section.js`

```

const SECTION_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      content
    }
  }
`

const withData = graphql(SECTION_QUERY, {
  options: { variables: { id: '1-1' } },
  props: ({ data: { section } }) => ({ section })
})

```

- `query SectionContent($id: String!) {` : We declare at the top that the `SectionContent` query takes a variable `$id`, a required `String`.
- `section(id: $id) {` : We replace our string literal `"1-1"` with the variable `$id`.
- `options: { variables: { id: '1-1' } }` : We tell `graphql()` to pass an `id` variable to the query.

Now passing the variable to the query is working, but we still have `'1-1'` hard-coded. Where do we get the section ID from? Back in `TableOfContents`, we gave a `to` prop to our `NavLink`:

```

<NavLink
  to={{
    pathname: slugify(chapter, section),
    state: { chapter, section }
  }}

```

The `pathname` is the equivalent of an anchor tag's `href` attribute, but `state` is part of the HTML5 [session history management](#) method). We can access it at `window.location.state`, but we also want our components to react to changes, so we want it as a prop. The best way to use browser history state with `react-router` is with the `withRouter` HOC, which provides our component with a `location` prop, which has a `.state` property. `graphql()`'s `options` can have a function value instead of our current `variables` object literal—the function takes the props and returns the variables:

`src/components/Section.js`

```
import { withRouter } from 'react-router'

const withData = graphql(SECTION_QUERY, {
  options: ({ location: { state: { section: { id } } } }) => ({
    variables: { id }
  }),
  props: ({ data: { section, loading } }) => ({
    sectionContent: section && section.content,
    loading
  })
})

export default withRouter(withData(Section))
```

If you get `TypeError: Cannot read property 'section' of undefined`, skip ahead to the next section to see the solution.

Our options function gets the section ID from the `location` prop and returns it as the query variable. In our `props` function, we change from passing `section` to just passing a `sectionContent` string (so that the name doesn't conflict with also getting the `section` from `location.state`). Also, our HOC order matters—we have to put `withRouter()` outside of `withData()` so that `graphql()` gets the `location` prop to give to the options function.

Let's fill in our component with our newly available data:

`src/components/Section.js`

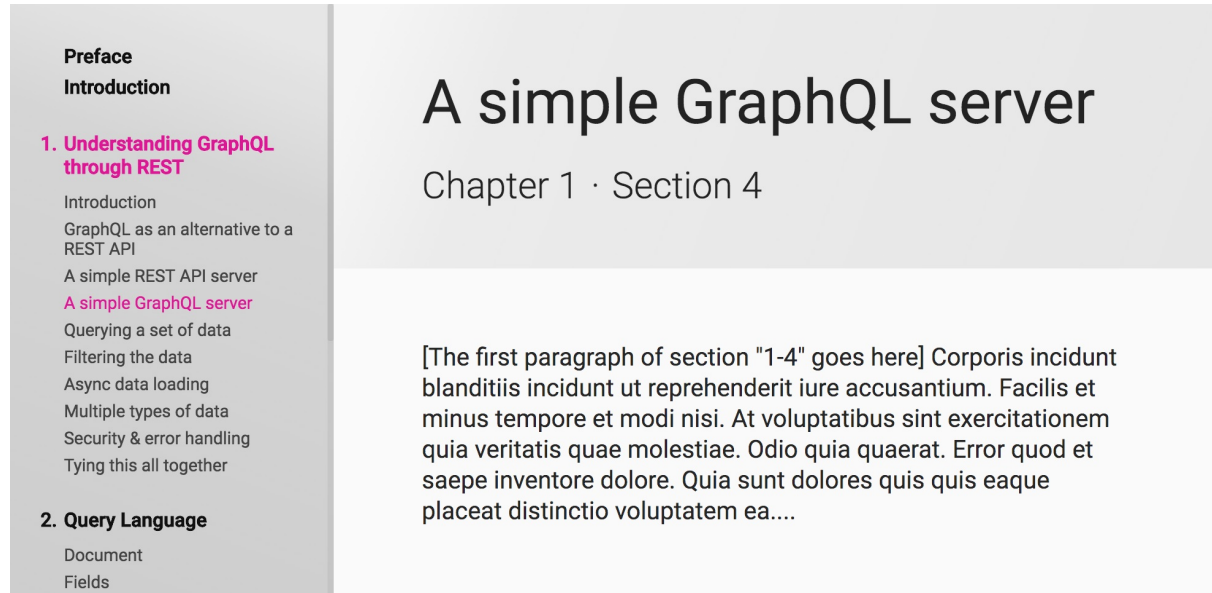
```
const Section = ({
  loading,
  sectionContent,
  location: { state: { chapter, section } }
}) => (
  <section className="Section">
    <div className="Section-header-wrapper">
      <header className="Section-header">
        {chapter.number !== null ? (
          <div>
            <h1>{section.title}</h1>
            <h2>
              {'Chapter ' + chapter.number}
              <span className="Section-number-divider" />
              {'Section ' + section.number}
            </h2>
          </div>
        ) : (
          <h1>{chapter.title}</h1>
        )}
      </header>
    </div>
    <div className="Section-content">
      {loading ? <Skeleton count={7} /> : sectionContent}
    </div>
  </section>
)

Section.propTypes = {
  sectionContent: PropTypes.string,
  location: PropTypes.object.isRequired,
  loading: PropTypes.bool.isRequired
}
```

```
}

```

We can see this working by clicking a different section in the table of contents. The path will change and a new `state` will be set, which `withRouter` will provide to `Section`, triggering a re-render, and the book content on the right will update.



## Skipping queries

If you're jumping in here, `git checkout 5_0.2.0` (tag [5\\_0.2.0](#), or compare [5...6](#))

If you've kept your development browser tab open during this section, then everything has worked smoothly for you. But when we open a [new tab](#), we find a bug:

```
TypeError: Cannot read property 'section' of undefined
```

```
const withData = graphql(SECTION_QUERY, {
  options: ({ location: { state: { section: { id } } } }) => ({
```

It looks like `location.state` is undefined! 🐛 Which makes sense, because in a new tab, we haven't yet clicked a `<NavLink>`, so the state hasn't been set. If we don't have the state, how do we get the section ID so that we can query for the right content? The only information we have on first page load is the path, so we have to parse it. `location.pathname` will always be defined, so we can `deslugify()` it:

[src/lib/helpers.js](#)

```
// parse a path:
// /Introduction
// -> { chapterTitle: 'Introduction' }
//
// /1-Understanding-GraphQL-through-REST/1-Introduction
// -> { chapterNumber: 1, sectionNumber: 1 }
export const deslugify = path => {
  const [, chapterSlug, sectionSlug] = path.split('/')
  const chapterIsNumbered = !!sectionSlug

  return chapterIsNumbered
    ? {
        chapterNumber: parseInt(chapterSlug.split('-')[0], 10),
        sectionNumber: parseInt(sectionSlug.split('-')[0], 10)
      }
    : { chapterTitle: chapterSlug }
```



```
}

```

Now let's look at Playground to figure out which two queries we can use, given either the chapter title or the chapter and section numbers:

Playground: `query { }`

We can use the `chapterByTitle` and `chapterByNumber` root query fields along with a `Chapter`'s `section` field with a `number: Int!` argument. (Any field, not just root fields, can have arguments.)

`src/components/Section.js`

```
const SECTION_BY_ID_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      content
    }
  }
`

const SECTION_BY_CHAPTER_TITLE_QUERY = gql`
  query SectionByChapterTitle($title: String!) {
    chapterByTitle(title: $title) {
      title
      section(number: 1) {
        content
      }
    }
  }
`

const SECTION_BY_NUMBER_QUERY = gql`
  query SectionByChapterTitle($chapterNumber: Int!, $sectionNumber: Int!) {
    chapterByNumber(number: $chapterNumber) {
      number
      section(number: $sectionNumber) {
        number
        title
        content
      }
    }
  }
`

```

For `chapterByTitle`, all the non-numbered chapters only have a single section and are numbered `0` and title-less. For the `chapterByNumber`, we need the section in addition to the contents, because we display it at the top of the component, and we no longer get it from `location.state`.

## Section HOCs

Now we need to figure out which query to use! We could make our own HOC that took in `location` from `withRouter` and chose which query to use, like this:

```
export default withRouter(withCorrectQuery(Section))

```

But let's instead try out the `skip` feature of `graphql()`. It allows you to provide a function that calculates from the props whether to perform the query. For instance, we don't want to use the `SECTION_BY_ID_QUERY` when there's no state:

```
const withSectionById = graphql(SECTION_BY_ID_QUERY, {
  skip: ({ location }) => !location.state,

```

Let's also standardize the props so that `Section` always gets a `section` and a `chapter`, with `section.content` :

src/components/Section.js

```
const SECTION_BY_ID_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      content
    }
  }
`

const withSectionById = graphql(SECTION_BY_ID_QUERY, {
  skip: ({ location }) => !location.state,
  options: ({ location: { state } }) => ({
    variables: { id: state && state.section.id }
  }),
  props: ({
    ownProps: { location: { state } },
    data: { section, loading }
  }) => ({
    section: {
      ...state.section,
      content: section && section.content
    },
    chapter: state.chapter,
    loading
  })
})
```

In our `props` function, we have access to `withRouter()`'s props under the `ownProps` argument. We add the section content we get from the query result to `ownProps.locations.state.section`. We need to guard against `data.section` being `undefined` ( `section && section.content` ) because it will be when `loading` is true.

Let's make an HOC for the next query:

src/components/Section.js

```
import { deslugify } from '../lib/helpers'

const SECTION_BY_CHAPTER_TITLE_QUERY = gql`
  query SectionByChapterTitle($title: String!) {
    chapterByTitle(title: $title) {
      title
      section(number: 1) {
        content
      }
    }
  }
`

const withSectionByChapterTitle = graphql(SECTION_BY_CHAPTER_TITLE_QUERY, {
  skip: ({ location }) =>
    location.state || !deslugify(location.pathname).chapterTitle,
  options: ({ location: { pathname } }) => ({
    variables: { title: deslugify(pathname).chapterTitle }
  }),
  props: ({ data: { chapterByTitle, loading } }) => ({
    section: chapterByTitle && chapterByTitle.section,
    chapter: {
      ...chapterByTitle,
      number: null
    },
    loading
  })
})
```

We want to skip over this query if we either have state (in which case we used `SECTION_BY_ID_QUERY`) or if the path doesn't have a `chapterTitle` (in which case we'll pass on to the next HOC/query). We get the `title` query variable from the path, and for props, `section` comes from the query results. We didn't need the `number` field in our query because we know that these chapters aren't numbered.

On to the next query 🐼:

`src/components/Section.js`

```
const SECTION_BY_NUMBER_QUERY = gql`
  query SectionByNumber($chapterNumber: Int!, $sectionNumber: Int!) {
    chapterByNumber(number: $chapterNumber) {
      number
      section(number: $sectionNumber) {
        number
        title
        content
      }
    }
  }
`

const withSectionByNumber = graphql(SECTION_BY_NUMBER_QUERY, {
  skip: ({ location }) =>
    location.state || !deslugify(location.pathname).chapterNumber,
  options: ({ location: { pathname } }) => ({ variables: deslugify(pathname) }),
  props: ({ data: { chapterByNumber, loading } }) => ({
    section: chapterByNumber && chapterByNumber.section,
    chapter: chapterByNumber,
    loading
  })
})
```

- `options`: What we get from `deslugify` in this case matches our query variable format (`{ chapterNumber: 1, sectionNumber: 1 }`).
- `props`: We get all the props we need from the query results.

We can combine all of our HOCs with:

```
export default withRouter(
  withSectionById(withSectionByChapterTitle(withSectionByNumber(Section)))
)
```

😬 but that's hard to read! We can do the same with `react-apollo`'s `compose()` (equivalent to `compose()` from `recompose` or `redux`):

```
import { graphql, compose } from 'react-apollo'

export default compose(
  withRouter,
  withSectionById,
  withSectionByChapterTitle,
  withSectionByNumber
)(Section)
```

All four HOCs get composed together to a single HOC, which is applied to `Section`. Here's the more verbose version of the same thing:

```
const withRouterAndData = compose(
  withRouter,
  withSectionById,
  withSectionByChapterTitle,
  withSectionByNumber
)
```

```
export default withRouterAndData(Section)
```

And we set our `PropTypes` :

```
Section.propTypes = {
  section: PropTypes.shape({
    title: PropTypes.string,
    number: PropTypes.number,
    content: PropTypes.string
  }),
  chapter: PropTypes.shape({
    title: PropTypes.string,
    number: PropTypes.number
  }).isRequired,
  loading: PropTypes.bool.isRequired
}
```

Now when we open </introduction> or </1-Understanding-GraphQL-through-REST/1-Introduction> in new tabs, we get the right section content instead of an error! 🐛👊

In [Apollo devtools](#), we can look at the active queries on the page, which will let us see which of our three `graphql()` HOCs is being used:

The first screenshot shows the Apollo DevTools interface with the 'Queries' tab selected. The 'Watched queries' list contains three items: 1. StarsQuery, 2. ChapterQuery, and 3. SectionContent. The 'SectionContent <Query>' query is expanded, showing its variables and query string.

The second screenshot shows the Apollo DevTools interface with the 'Queries' tab selected. The 'Watched queries' list contains three items: 1. StarsQuery, 2. ChapterQuery, and 3. SectionByNumber. The 'SectionByNumber <Query>' query is expanded, showing its variables and query string.

**SectionContent <Query>** Run in GraphQL

**Variables**

- id "5-1"

**Query string**

```
query SectionContent($id: String!) {
  section(id: $id) {
    content
    __typename
  }
}
```

**SectionByNumber <Query>** Run in GraphQL

**Variables**

- chapterNumber 5
- sectionNumber 1

**Query string**

```
query SectionByNumber($chapterNumber: Int!, $sectionNumber: Int!) {
  chapterByNumber(number: $chapterNumber) {
    number
    section(number: $sectionNumber) {
      number
      title
      content
      __typename
    }
    __typename
  }
}
```

The first image is from a tab in which we've been navigating with the table of contents, and it uses the `SectionContent` query. The second image is from a newly opened tab, and it uses `SectionByNumber`.

## Section Query

As we learned in `StarCount.js`, a benefit to the `<Query>` component is being able to dynamically decide on a query to use. That's a perfect fit for `Section.js`, where we're deciding on one of three queries to use. Here are our three HOCs replaced with a `<Query>`:

`src/components/Section.js`

```
import { Query } from 'react-apollo'

const SectionWithData = ({ location: { state, pathname } }) => {
  const page = deslugify(pathname)

  let query, variables

  if (state) {
    query = SECTION_BY_ID_QUERY
    variables = { id: state.section.id }
  } else if (page.chapterTitle) {
    query = SECTION_BY_CHAPTER_TITLE_QUERY
    variables = { title: page.chapterTitle }
  } else if (page.chapterNumber) {
    query = SECTION_BY_NUMBER_QUERY
    variables = page
  }

  return (
    <Query query={query} variables={variables}>
      {{{ data, loading }}} => {
        let props

        if (data.section) {
          props = {
            section: {
              ...state.section,
              content: data.section.content
            },
            chapter: state.chapter,
            loading
          }
        } else if (data.chapterByTitle) {
          props = {
            section: data.chapterByTitle.section,
            chapter: {
              ...data.chapterByTitle,
              number: null
            },
            loading
          }
        } else if (data.chapterByNumber) {
          props = {
            section: data.chapterByNumber.section,
            chapter: data.chapterByNumber,
            loading
          }
        } else {
          props = { loading }
        }

        return <Section {...props} />
      }}}
    </Query>
  )
}

export default withRouter(SectionWithData)
```

First we decide, based on the `location` prop, which `query` and `variables` to use. Then inside the render prop, `data` will have either a `section`, `chapterByTitle`, or `chapterByNumber` attribute, depending on which query was used. Based on which data is returned, we can construct the right props for `<Section>`. Let's compare to the HOC solution:

```

const withSectionById = graphql(SECTION_BY_ID_QUERY, {
  skip: ({ location }) => !location.state,
  options: ({ location: { state } }) => ({
    variables: { id: state && state.section.id }
  }),
  props: ({
    ownProps: { location: { state } },
    data: { section, loading }
  }) => ({
    section: {
      ...state.section,
      content: section && section.content
    },
    chapter: state.chapter,
    loading
  })
})

const withSectionByChapterTitle = graphql(SECTION_BY_CHAPTER_TITLE_QUERY, {
  skip: ({ location }) =>
    location.state || !deslugify(location.pathname).chapterTitle,
  options: ({ location: { pathname } }) => ({
    variables: { title: deslugify(pathname).chapterTitle }
  }),
  props: ({ data: { chapterByTitle, loading } }) => ({
    section: chapterByTitle && chapterByTitle.section,
    chapter: {
      ...chapterByTitle,
      number: null
    },
    loading
  })
})

const withSectionByNumber = graphql(SECTION_BY_NUMBER_QUERY, {
  skip: ({ location }) =>
    location.pathname || !deslugify(location.pathname).chapterNumber,
  options: ({ location: { pathname } }) => ({ variables: deslugify(pathname) }),
  props: ({ data: { chapterByNumber, loading } }) => ({
    section: chapterByNumber && chapterByNumber.section,
    chapter: chapterByNumber,
    loading
  })
})

export default compose(
  withRouter,
  withSectionById,
  withSectionByChapterTitle,
  withSectionByNumber
)(Section)

```

Instead of an if-else statement, the HOC solution uses `skip`. It also has to call `deslugify` and deconstruct arguments more often. A downside of our `<Query>` implementation is that the query and variables are separated from the prop creation, but we can fix that by using a function:

```

import get from 'lodash/get'

const SectionWithData = ({ location: { state, pathname } }) => {
  const page = deslugify(pathname)

  let query, variables, createProps

  if (state) {
    query = SECTION_BY_ID_QUERY
    variables = { id: state.section.id }
    createProps = ({ data, loading }) => ({
      section: {
        ...state.section,
        content: get(data, 'section.content')
      },

```

```

        chapter: state.chapter,
        loading
      })
    } else if (page.chapterTitle) {
      query = SECTION_BY_CHAPTER_TITLE_QUERY
      variables = { title: page.chapterTitle }
      createProps = ({ data, loading }) => ({
        section: get(data, 'chapterByTitle.section'),
        chapter: {
          ...data.chapterByTitle,
          number: null
        },
        loading
      })
    } else if (page.chapterNumber) {
      query = SECTION_BY_NUMBER_QUERY
      variables = page
      createProps = ({ data, loading }) => ({
        section: get(data, 'chapterByNumber.section'),
        chapter: data.chapterByNumber,
        loading
      })
    }
  }

  return (
    <Query query={query} variables={variables}>
      {queryInfo => <Section {...createProps(queryInfo)} />}
    </Query>
  )
}

export default withRouter(SectionWithData)

```

Since, for example, `data.section` might be undefined, we have to either go back to guarding (`data.section && data.section.content`) or use `lodash/get`.

Let's go with the `<Query>` version, since the logic of which query should be used is more clear, and being able to easily understand what's happening (from the point of view of a teammate reading for the first time, or—let's be honest—ourselves looking back at it a month later 😊) is one of the most important factors of code quality.

## More routing

Before we move on to the authentication section, we've got another bug! You may very well have already noticed this one—we can't visit the root URL `localhost:3000/` 🤔. `TypeError: Cannot read property 'number' of undefined` is coming from our `chapter.number !== null` check in `Section`, and `chapter` is undefined because *none* of our HOCs was able to fetch the chapter. That's because on the root, there's neither history state nor a path to `deserialize()`. While it would be nice to redirect from the root to whichever section the user was last reading, for now let's just redirect to the first chapter.

So far, we haven't defined any routes—`Section` just changes what data it shows based on the path. We can create a root route that redirects to `/Preface`.

`src/components/App.js`

```

import { Switch, Route, Redirect } from 'react-router'

const Book = () => (
  <div>
    <TableOfContents />
    <Section />
  </div>
)

class App extends Component {
  render() {
    return (

```

```

    <div className="App">
      <header className="App-header">
        <StarCount />
        <img src={logo} className="App-logo" alt="logo" />
        <h1 className="App-title">The GraphQL Guide</h1>
      </header>
      <Switch>
        <Route exact path="/" render={() => <Redirect to="/Preface" />} />
        <Route component={Book} />
      </Switch>
    </div>
  )
}
}

```

Assuming we always want to keep our header on the page regardless of which route we're on, we put the `<Route>`s below the header in lieu of `<TableOfContents />` and `<Section />`, which we move to a new `Book` component.

`<Switch>` renders the first `<Route>` that matches. The first route matches only `/` and redirects, and the second route matches everything else and displays `Book`.

This begs the question, "What happens when the second route matches `/aMistypedChapterTitle` "? We get another error! This time it's `TypeError: Cannot read property 'content' of null` from `Section` trying to display `section.content`, because we failed at fetching `section`. Let's have `Section` display a 404 message, and let's refactor, since the double ternary operator is already hard to read.

`src/components/Section.js`

```

const Section = ({ loading, section, chapter }) => {
  let headerContent = null,
      sectionContent = null

  if (loading) {
    headerContent = (
      <h1>
        <Skeleton />
      </h1>
    )
    sectionContent = <Skeleton count={7} />
  } else if (!section) {
    headerContent = <h1> 404 page not found</h1>
  } else {
    if (chapter.number !== null) {
      headerContent = (
        <div>
          <h1>{section.title}</h1>
          <h2>
            {'Chapter ' + chapter.number}
            <span className="Section-number-divider" />
            {'Section ' + section.number}
          </h2>
        </div>
      )
    } else {
      headerContent = <h1>{chapter.title}</h1>
    }

    sectionContent = section.content
  }

  return (
    <section className="Section">
      <div className="Section-header-wrapper">
        <header className="Section-header">{headerContent}</header>
      </div>
      <div className="Section-content">{sectionContent}</div>
    </section>
  )
}

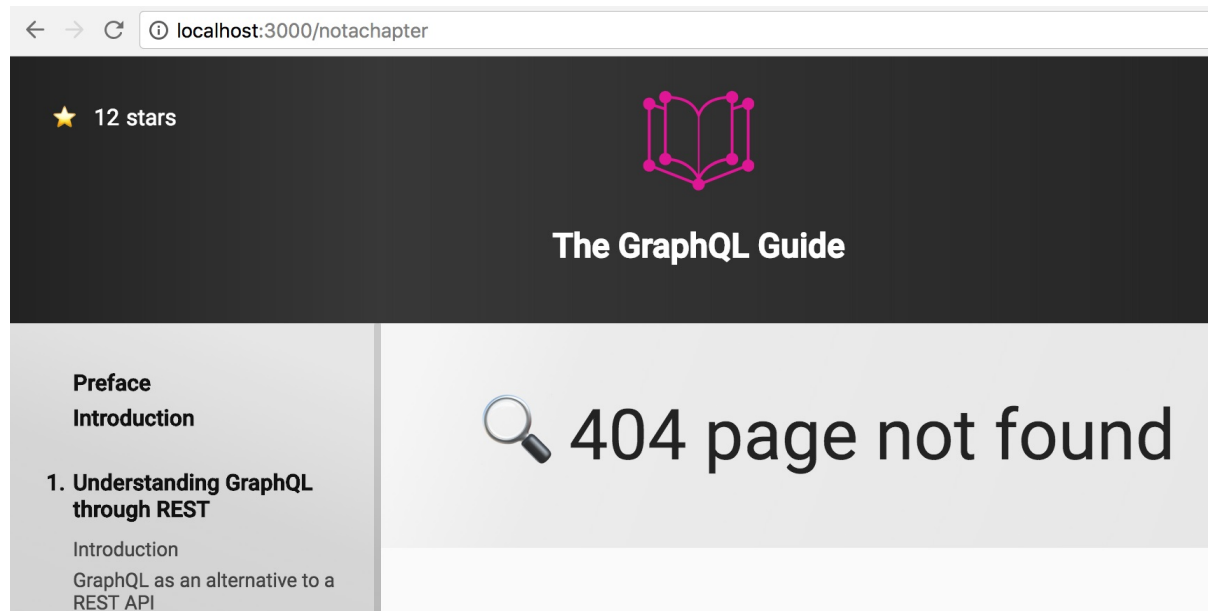
```



First we check if we're loading (in which case we don't know whether we've failed to find `section` yet), then we check if we didn't find `section`. If neither of those cases applied, then we would render the title and content.

Now we should be able to both:

- go to the root [localhost:3000/](http://localhost:3000/) and get redirected
- go to [/notachapter](http://localhost:3000/notachapter) and see the 404 message:



## Authentication

Section contents:

- [Logging in](#)
- [Resetting](#)

## Logging in

Background: [Authentication](#)

If you're jumping in here, `git checkout 6_0.2.0` (tag [6\\_0.2.0](#), or compare [6...7](#))

We'll have noticed by now that we're not getting the entire section content from the Guide API, and that's because we're not logged in. When we bought the book, we created a user account that was associated with our purchase. In order to see the full content, we need to log in with that account.

Authentication is important and complex enough that we rarely want to code it ourselves—we probably should use a library or service. For node backends, the most common library is [passport](#). We'll instead use a service—[Auth0](#)—for ease of integration. There are pros and cons to [signed tokens vs. sessions](#) and [localStorage vs. cookies](#), but we'll go with the most straightforward option for Auth0 integration: tokens stored in `localStorage`. They have a number of authentication methods (called "Connections" in Auth0 or "strategies" in Passport), including email/password, [passwordless](#) (SMS one-time codes, email magic login links, and/or TouchID), and Social OAuth providers. While Auth0 makes it easy to provide multiple options, for simplicity's sake, we'll just provide GitHub OAuth—all of our users are developers, and they're likely already logged into their GitHub account on most of their browsers, so the login process should be really easy. If we were building for a different market, we might prefer passwordless instead.

A common login sequence is this: the user clicks a login button, which redirects them to the GitHub OAuth page, and after they do GitHub login (if needed), they authorize our app and are redirected back to our site. One UX drawback of this sequence is that at the end, the user has to wait for our site to load, and without some work, they won't be taken to the exact page and scroll position they were at before. A good alternative is to open a popup (or a new tab on mobile) where the user can do the GitHub steps. When they're done authorizing, the popup closes and returns the signed token to the app. Then we'll include that token in our requests to the server so the server will know who the user is.

Let's think about what UI elements we want related to the login and the user. We can put a login link on the right side of the header, which will open the GitHub popup. Once the user is logged in, we can show their GitHub profile photo and name in place of the login link, and if they click their name, we can take them to a new `/me` route that shows them their profile. For all of this, we'll need some data and functions—the user data, whether the user data is loading, and login and logout functions. We need it in a couple of different places in the app—in the header and in a route. There are a few different ways to get information to any place in the app—one is to render an `<AppContainer>` instead of `<App>` in `index.js`:

```
ReactDOM.render(
  <BrowserRouter>
    <ApolloProvider client={client}>
      <AppContainer />
    </ApolloProvider>
  </BrowserRouter>,
  document.getElementById('root')
)
```

And then the `<AppContainer>` fetches the current user object from the server and passes it to `<App>` along with login/logout functions and `loggingIn`—whether the app is in the process of logging the user in:

```
class AppContainer extends Component {
  render() {
    ...
    return (
      <App
        user={user}
        login={this.login}
        logout={this.logout}
        loggingIn={loading}
      />
    )
  }
}
```

Then `<App>` in turn passes the props down the component tree to children and grandchildren who need them. The main benefit to this method is that it's easy to test, because it's simple to mock out props. However, in all but the smallest apps, it results in a lot of *prop drilling* (passing props down to a component's children's children's ... children). That can get tiresome and clutter our JSX and PropTypes. Instead, let's make `login()` and `logout()` global functions and let's use `graphql()` to create a `withUser()` HOC that provides `user` and `loggingIn`. Then inside components that deal the with user, we can import and use `login()`, `logout()`, and `withUser()`.

Let's add the current user's name and photo to our header, and let's add a route for a profile page:

`src/components/App.js`

```
import { Link } from 'react-router-dom'

import CurrentUser from '../CurrentUser'
import Profile from '../Profile'

class App extends Component {
  render() {
    return (
```

```

    <div className="App">
      <header className="App-header">
        <StarCount />
        <Link className="App-home-link" to="/">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">The GraphQL Guide</h1>
        </Link>
        <CurrentUser />
      </header>
      <Switch>
        <Route exact path="/" render={() => <Redirect to="/Preface" />} />
        <Route exact path="/me" component={Profile} />
        <Route component={Book} />
      </Switch>
    </div>
  )
}
}

```

We call the header component `<CurrentUser>` because that's what it will usually be displaying (it will sometimes instead have a "Sign in" button or a spinner). We need a way for the user to navigate from `/me` to the rest of the app, so we wrap the header image and title in a `<Link>` to the root `/`. Later we'll get to the HOC ( `lib/withUser.js` ) and the login/logout functions ( `lib/auth.js` ), but for now let's assume they work and write `<CurrentUser>` :

`src/components/CurrentUser.js`

```

import React from 'react'
import PropTypes from 'prop-types'
import { Link } from 'react-router-dom'

import { withUser } from '../lib/withUser'
import { login } from '../lib/auth'

const CurrentUser = ({ user, loggingIn }) => {
  let content

  if (user) {
    content = (
      <Link to="/me" className="User">
        <img src={user.photo} alt={user.firstName} />
        {user.firstName}
      </Link>
    )
  } else if (loggingIn) {
    content = <div className="Spinner" />
  } else {
    content = <button onClick={login}>Sign in</button>
  }

  return <div className="CurrentUser">{content}</div>
}

CurrentUser.propTypes = {
  user: PropTypes.shape({
    firstName: PropTypes.string.isRequired,
    photo: PropTypes.string.isRequired
  }),
  loggingIn: PropTypes.bool.isRequired
}

export default withUser(CurrentUser)

```

This one is straightforward to read. If there's no user and the user isn't being loaded, then we have a "Sign in" button that calls `login()` .

Similarly, in `<Profile>` , we might show a loading spinner or a login button. Otherwise, we show the user's details and a "Sign out" button:

`src/components/Profile.js`

```

import React from 'react'
import PropTypes from 'prop-types'

import { withUser } from '../lib/withUser'
import { login, logout } from '../lib/auth'

const Profile = ({ user, loggingIn }) => {
  if (loggingIn) {
    return (
      <main className="Profile">
        <div className="Spinner" />
      </main>
    )
  } else if (!user) {
    return (
      <main className="Profile">
        <button onClick={login} className="Profile-login">
          Sign in
        </button>
      </main>
    )
  } else {
    return (
      <main className="Profile">
        <div className="Profile-header-wrapper">
          <header className="Profile-header">
            <h1>{user.name}</h1>
          </header>
        </div>
        <div className="Profile-content">
          <dl>
            <dt>Email</dt>
            <dd>
              <code>{user.email}</code>
            </dd>

            <dt>Membership level</dt>
            <dd>
              <code>{user.hasPurchased || 'GUEST'}</code>
            </dd>

            <dt>OAuth Github account</dt>
            <dd>
              <a
                href="https://github.com/settings/applications"
                target="_blank"
                rel="noopener noreferrer"
              >
                <code>{user.username}</code>
              </a>
            </dd>
          </dl>

          <button className="Profile-logout" onClick={logout}>
            Sign out
          </button>
        </div>
      </main>
    )
  }
}

Profile.propTypes = {
  user: PropTypes.shape({
    name: PropTypes.string.isRequired,
    email: PropTypes.string.isRequired,
    hasPurchased: PropTypes.string
  }),
  loggingIn: PropTypes.bool.isRequired
}

export default withUser(Profile)

```

And now to write our authentication logic! First, we need to set up the Auth0 client:

src/lib/auth.js

```
import auth0 from 'auth0-js'
import {
  initAuthHelpers,
  login as auth0Login,
  logout as auth0Logout
} from 'auth0-helpers'

const client = new auth0.WebAuth({
  domain: 'graphql.auth0.com',
  clientId: '8fErnZoF3hbzQ2AbMYu5xcS0aVNzQ0PC',
  responseType: 'token',
  audience: 'https://api.graphql.guide',
  scope: 'openid profile guide'
})

initAuthHelpers({
  client,
  usePopup: true,
  auth0Options: {
    connection: 'github',
    owp: true,
    popupOptions: { height: 623 } // make tall enough for content
  },
  checkSessionOptions: {
    redirect_uri: window.location.origin
  },
  onError: e => console.error(e)
})
```

Here we're just following the docs for `auth0-js` and `auth0-helpers`. Now `auth0Login()` and `auth0Logout()` should be configured to work with the Guide's Auth0 account system, and we can use them:

src/lib/auth.js

```
export const login = () => {
  auth0Login({
    onCompleted: e => {
      if (e) {
        console.error(e)
        return
      }
    }
  })
}

export const logout = () => {
  auth0Logout()
}
```

You might be wondering, "But what do the login and logout functions actually do?" `auth0Login()` opens the GitHub auth popup, and saves the resulting token in `localStorage`. `auth0Logout()` removes the token from `localStorage` and ends our session with the Auth0 server. The next step is actually using the token—whenever we communicate with the server, we need to provide it. There's an Apollo Link called `setContext` that lets us set headers on HTTP requests, and we'll use it to add an `authorization` header with the token. While we're at it, let's move our Apollo client creation out to another file:

src/index.js

```
import { apollo } from './lib/apollo'

ReactDOM.render(
  <BrowserRouter>
    <ApolloProvider client={apollo}>
```

```

    <App />
  </ApolloProvider>
</BrowserRouter>,
document.getElementById('root')
)

```

src/lib/apollo.js

```

import { ApolloClient } from 'apollo-client'
import { InMemoryCache } from 'apollo-cache-inmemory'
import { split } from 'apollo-link'
import { WebSocketLink } from 'apollo-link-ws'
import { createHttpLink } from 'apollo-link-http'
import { getMainDefinition } from 'apollo-utilities'
import { setContext } from 'apollo-link-context'
import { getAuthToken } from 'auth0-helpers'

const httpLink = createHttpLink({
  uri: 'https://api.graphql.guide/graphql'
})

const authLink = setContext(async (_, { headers }) => {
  const token = await getAuthToken({
    doLoginIfTokenExpired: true
  })

  if (token) {
    return {
      headers: {
        ...headers,
        authorization: `Bearer ${token}`
      }
    }
  } else {
    return { headers }
  }
})

const authedHttpLink = authLink.concat(httpLink)

const wsLink = new WebSocketLink({
  uri: `wss://api.graphql.guide/subscriptions`,
  options: {
    reconnect: true
  }
})

const link = split(
  ({ query }) => {
    const { kind, operation } = getMainDefinition(query)
    return kind === 'OperationDefinition' && operation === 'subscription'
  },
  wsLink,
  authedHttpLink
)

const cache = new InMemoryCache()

export const apollo = new ApolloClient({ link, cache })

```

We get the token from `auth0-helpers` using `getAuthToken()`, which either looks it up in `localStorage`, or if it has expired, opens the GitHub auth popup again. We use `concat()` to combine our new `authLink` with the `httpLink` — now when our Apollo client sends out a new query or mutation, it will first go through `authLink`, which will set the header, and then through `httpLink`, which will put it in an HTTP request and send it to the server.

The last piece is to make an HOC that provides the current user's data:

src/lib/withUser.js

```
import { graphql } from 'react-apollo'
import gql from 'graphql-tag'

export const USER_QUERY = gql`
  query UserQuery {
    currentUser {
      id
      firstName
      name
      username
      email
      photo
      hasPurchased
    }
  }
`

export const withUser = graphql(USER_QUERY, {
  props: ({ data: { currentUser, loading } }) => ({
    user: currentUser,
    loggingIn: loading
  })
})
```

We can now try logging in with our Github account. Clicking sign in opens the popup, and after we go through the OAuth dialog, the popup closes. But then nothing else happens. The “Sign in” link is still there, which means `withUser()` is still providing `user: null` to `<CurrentUser>`. If we reload, it'll show us logged in, but we don't want to have to reload, of course. This issue will be solved in the next section.

## Resetting

If you're jumping in here, `git checkout 7_0.2.0` (tag `7_0.2.0`, or compare `7...8`)

Because the auth token is included in every request, the server will know who we are for any other queries and mutations we send, like the ones for the section content. So our server should recognize that we have purchased a Guide package and return the full content to the sections that are included in our package. But after we log in, the section content is still cut off like it was before. Why is that? Because the section content queries haven't been refetched! We're still showing the old data fetched when we were logged out. Now what do we do?

Apollo does have a `refetch()` function that we get along with a query's results. It would be a pain to use on our section queries because: A) there are 3 of them, and B) we'd have to figure out how to call the `refetch()` functions (which would be inside `Section.js`) from `auth.js`. So let's take a different path—telling Apollo to refetch all the queries in the app. Apollo has a `refetchObservableQueries()` function, which takes all the *observable queries* (queries used in a `<Query>` or `graphql()` to provide data to our components) and re-sends them to the server. Let's call that:

`src/lib/auth.js`

```
import { apollo } from './apollo'

export const login = () => {
  auth0Login({
    onCompleted: e => {
      if (e) {
        console.error(e)
        return
      }

      apollo.refetchObservableQueries()
    }
  })
}
```

Now we've got login working. But let's take a minute to think about query efficiency. We're using `withUser()` twice right now, and when we load `/me`, it's used two places on the page. But if we look in our network tab, we only see `UserQuery` sent to the server once! This is an example of Apollo's automatic [query deduplication](#)—when we ask it to make the same query twice, it's smart enough to only send it once and give the result to both components. However, whenever we render new components that use `withUser()` (for instance, when we navigate from `/Preface` to `/me`), it's treated as a separate query and not deduplicated. But we don't need to re-send it to the server—the user's name, photo, etc. isn't likely to change. Luckily, it isn't! The default `fetchPolicy` for queries is `cache-first`, which means if the query result is already in the cache, Apollo loads the data from the cache. If we were dealing with a type of data that was more likely to change, we could set the `fetchPolicy` to `cache-and-network`, which first loads data from the cache, but at the same time sends the query to the server, and will update the component if the server result is different from the cache result. We would set `fetchPolicy` like this:

```
export const withUser = graphql(USER_QUERY, {
  options: { fetchPolicy: 'cache-and-network' },
```

So our queries update on login, but what about logout? There may be private data in the store, so the method we want is `resetStore()`, which first clears the store and then refetches observable queries:

`src/lib/auth.js`

```
export const logout = () => {
  auth0Logout()
  apollo.resetStore()
}
```

Now when we log in and out, the full section content should appear and disappear.

## Mutating

Section contents:

- [First mutation](#)
- [Listing reviews](#)
- [Optimistic updates](#)
- [Arbitrary updates](#)
- [Creating reviews](#)
- [Using fragments](#)
- [Deleting](#)
- [Error handling](#)
- [Editing reviews](#)

## First mutation

If you're jumping in here, `git checkout 8_0.2.0` (tag [8\\_0.2.0](#), or compare [8...9](#))

We haven't yet changed any of the data in the Guide's database (just the star count in GitHub's database). When we want to change data (or more broadly, trigger side effects), we need to send a mutation to the server. Let's start with something simple—at the bottom of a `<Section>`, let's add the count of how many times the current section has been viewed. Then we can increment the count whenever it's viewed.

First we add the `views` field to each of our three section queries. Here's the first one:

`src/components/Section.js`



```
const SECTION_BY_ID_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      id
      content
      views
    }
  }
`

const SectionWithData = ({ location: { state, pathname } }) => {
  const page = deslugify(pathname)

  let query, variables, createProps

  if (state) {
    query = SECTION_BY_ID_QUERY
    variables = { id: state.section.id }
    createProps = ({ data, loading }) => ({
      section: {
        ...state.section,
        content: get(data, 'section.content'),
        views: get(data, 'section.views')
      },
      chapter: state.chapter,
      loading
    })
  } ...
}
```

In addition to `views`, we have to add `id` to the query's selection set so that the `Section` gets [normalized](#) correctly. Also, for this query, we need to add `get(data, 'section.views')` to `createProps()`. We don't have to modify the other `createProps()` functions, as they include the whole section instead of putting it together from different places.

Next we display the new data:

`src/components/Section.js`

```
let headerContent = null,
    sectionContent = null,
    footerContent = null

if (loading) {
  ...
} else if (!section) {
  ...
} else {
  ...

  sectionContent = section.content
  footerContent = `Viewed ${section.views.toLocaleString()} times`
}

return (
  <section className="Section">
    ...
    <footer>{footerContent}</footer>
  </section>
)
```

## Preface

### Introduction

#### 1. Understanding GraphQL through REST

##### Introduction

GraphQL as an alternative to a REST API

A simple REST API server

A simple GraphQL server

Querying a set of data

Filtering the data

Async data loading

Multiple types of data

Security & error handling

Tying this all together

#### 2. Query Language

Document

Fields

Arguments

Fragments

Variables

Directives

Mutations

Subscriptions

# Introduction

## Chapter 1 · Section 1

[The first paragraph of section "1-1" goes here] Et consequatur qui sapiente commodi. Dolor sint ut nemo ut exercitationem corporis est et optio. A nesciunt voluptas dolore aliquam omnis aut ea. Amet repudiandae odit nulla voluptates sunt veritatis consequatur rerum. Provident saepe eum sunt impedit sunt commodi aut in dolorem. Consequuntur eligendi nesciunt repudiandae....

Viewed 5 times

Now look for the mutation we need in Playground—we need the name, arguments, and return type.

Playground: `mutation { }`

And we write out the mutation string just like we write queries:

src/components/Section.js

```
const VIEWED_SECTION_MUTATION = gql`
  mutation ViewedSection($id: String!) {
    viewedSection(id: $id) {
      id
      views
    }
  }
`
```

Like in the queries, we need the `id` field so that Apollo knows which `Section` is being returned in the mutation response. Now the response's `views` field will update the normalized `Section` object in the Apollo store, which will update any component queries that select that field. Those queries will pass the updated info to the render prop—in this case, `withSectionById()` will pass a new `data.section` argument to the render prop. We'll be able to see this in action in a bit.

The mutation HOC is simpler than our query HOCs, since we don't have props to pass down:

src/components/Section.js

```
import { graphql } from 'react-apollo'

const SectionWithMutation = graphql(VIEWED_SECTION_MUTATION, {
  name: 'viewedSection'
})(Section)
```

```
...
<Query query={query} variables={variables}>
  {queryInfo => <SectionWithMutation {...createProps(queryInfo)} />}
</Query>
```

For mutations, `graphql()` creates an HOC that provides a single prop function, which we're naming `viewedSection`. We want to call it whenever a section is viewed, so inside `componentDidMount()` and `componentDidUpdate()`. In order to get lifecycle methods, we need to convert our functional component into a class:

`src/components/Section.js`

```
class Section extends Component {
  viewedSection = id => {
    if (!id) {
      return
    }

    this.timeoutID = setTimeout(() => {
      this.props.viewedSection({
        variables: { id }
      })
    }, 2000)
  }

  componentDidMount() {
    this.viewedSection(get(this, 'props.section.id'))
  }

  componentWillUnmount() {
    clearTimeout(this.timeoutID)
  }

  componentDidUpdate(prevProps) {
    const { id } = this.props.section
    const sectionChanged = get(prevProps, 'section.id') !== id

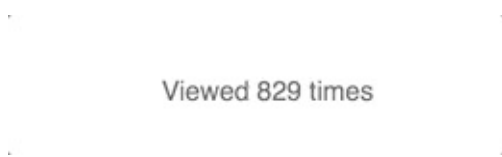
    if (sectionChanged) {
      this.viewedSection(id)
    }
  }

  render() { ... }
}

Section.propTypes = {
  ...
  viewedSection: PropTypes.func.isRequired
}
```

We give `this.props.viewedSection()` the section ID mutation variable. We put it in a timeout so that we have time to scroll down to the bottom of the section to see the count change (End key or Cmd-:arrow\_down: on Mac). And we clear the timeout on unmount (because if we navigate away, for example to our profile, and our timeout still fires, it would call a mutation provided by a `<Mutation>` component that no longer existed, and React would throw an error).

We also need to only trigger the mutation when the section changed. When the mutation result arrives and updates the Apollo store, `<Section>` is going to be given the updated `section` prop, so `componentDidUpdate()` will be called again. And if it always called `viewedSection()`, we'd be in an infinite loop. (Read: author Loren was stuck in an infinite loop 😅.)



We should now be able to see the count change at the bottom of the page when we switch between sections.

There is also a render prop API for mutations. Let's see what that looks like:

`src/components/Section.js`

```
import { Mutation } from 'react-apollo'

...

<Query query={query} variables={variables}>
  {queryInfo => (
    <Mutation mutation={VIEWED_SECTION_MUTATION}>
      {viewedSection => (
        <Section
          {...createProps(queryInfo)}
          viewedSection={viewedSection}
        />
      )}
    </Mutation>
  )}
</Query>
```

The render prop is given the mutation function, and we pass it to `<Section>`. We begin to see here how the indentation level can balloon when a component needs multiple queries and mutations—we add 2+ levels per operation (only 2 if we use implicit-return arrow functions, 3+ for functions with [blocks](#)).

## Listing reviews

If you're jumping in here, `git checkout 9_0.2.0` (tag [9\\_0.2.0](#), or compare [9...10](#))

Before we get to more advanced mutations, we need more stuff to work with! Let's make a new page that lists book reviews, and then in the [next section](#), we can implement features that require mutations: favoriting reviews, creating new reviews, and editing and deleting our own reviews.

Let's start out by adding a link to the bottom of the table of contents:

`src/components/TableOfContents.js`

```
const TableOfContents = ({ chapters, loading }) => (
  <nav className="TableOfContents">
    ...
    <li>
      <NavLink className="TableOfContents-reviews-link" to="/reviews">
        Reviews
      </NavLink>
    </li>
  </ul>
)>
</nav>
)
```

And we can add the new route with another `<Switch>`:

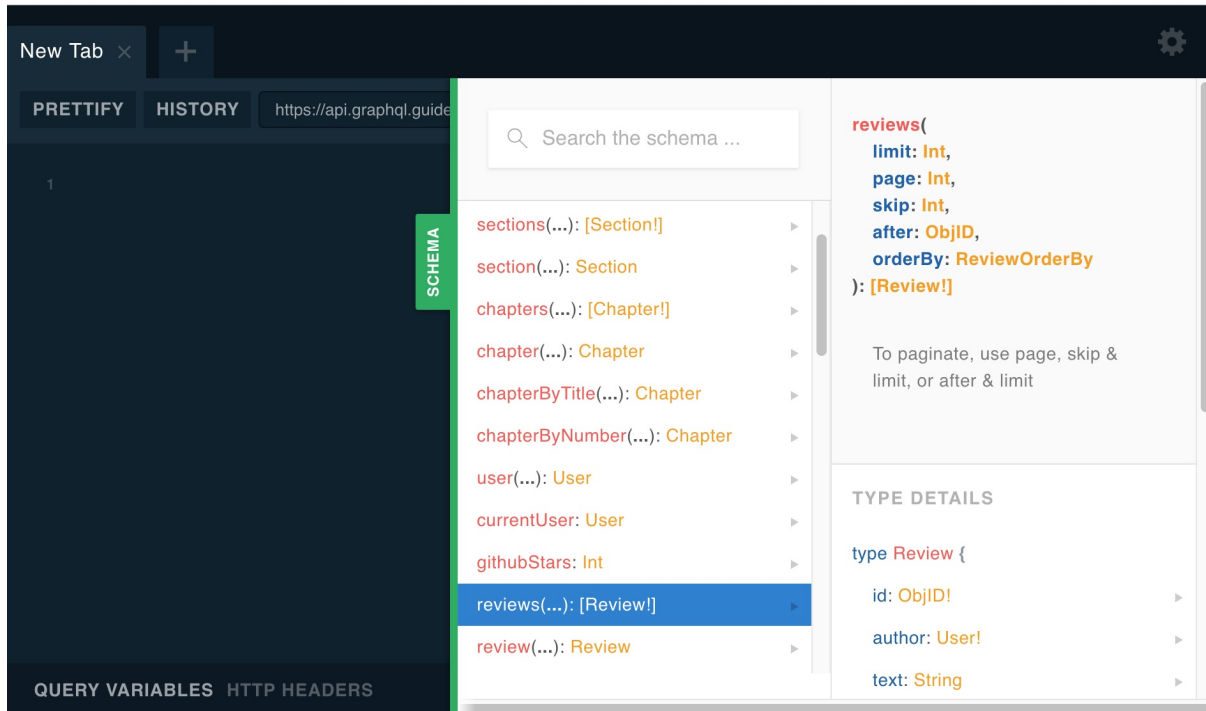
`src/components/App.js`

```
const Book = () => (
  <div>
    <TableOfContents />
    <Switch>
      <Route exact path="/reviews" component={Reviews} />
      <Route component={Section} />
    </Switch>
  </div>
)
```

Our `<Reviews>` component is going to need some data! We know how to do that now. Let's search through the schema for the right query:

Playground: `query { }`

We find the `reviews` root query field, and since fetching them all might be a lot of data, let's use the `limit` argument.



And for each review, we want to display the author's name, photo, and a link to their GitHub, so we need:

`src/components/Reviews.js`

```
const REVIEWS_QUERY = gql`
  query ReviewsQuery {
    reviews(limit: 20) {
      id
      text
      stars
      createdAt
      favorited
      author {
        id
        name
        photo
        username
      }
    }
  }
`
```

As before, we will use `graphql()` to get `reviews` and `loading` passed as props, and it should have a similar structure to `<Section>`:

`src/components/Reviews.js`

```
import React from 'react'
import PropTypes from 'prop-types'
import { graphql } from 'react-apollo'
import gql from 'graphql-tag'
```

```
import Review from './Review'

const Reviews = ({ reviews, loading }) => (
  <main className="Reviews mui-fixed">
    <div className="Reviews-header-wrapper">
      <header className="Reviews-header">
        <h1>Reviews</h1>
      </header>
    </div>
    <div className="Reviews-content">
      {loading ? (
        <div className="Spinner" />
      ) : (
        reviews.map(review => <Review key={review.id} review={review} />)
      )}
    </div>
  </main>
)

Reviews.propTypes = {
  reviews: PropTypes.arrayOf(PropTypes.object),
  loading: PropTypes.bool.isRequired
}

const withReviews = graphql(REVIEWS_QUERY, {
  props: ({ data: { reviews, loading } }) => ({ reviews, loading })
})

export default withReviews(Reviews)
```

We can be vague here with `reviews: PropTypes.arrayOf(PropTypes.object)` since we're not using individual `review` objects in this component. In `<Review>`, we'll list out the `review` fields used with a `PropTypes.shape`.

Next up is the `<Review>` component. So far we've mostly been using plain HTML tags and CSS classes for styling. For many components of an app, it's easier to use a library instead of building and styling them ourselves. The most popular React component library right now is [Material-UI](#), based on Google's [design system](#).

Here are some of the other [major React component libraries](#).

We can explore their component demos to find components we want to use to make up a `<Review>`, and we can browse the [material icons listing](#). Let's put each review on a `Card`, with an `Avatar` for the author's photo, a `MoreVert` and `Menu` for editing and deleting, and a more prominent `FavoriteBorder` as a bottom action:

`src/components/Reviews.js`

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import {
  Card,
  CardHeader,
  CardContent,
  CardActions,
  IconButton,
  Typography,
  Avatar,
  Menu,
  MenuItem
} from '@material-ui/core'
import {
  MoreVert,
  Favorite,
  FavoriteBorder,
  Star,
  StarBorder
} from '@material-ui/icons'
import distanceInWordsToNow from 'date-fns/distance_in_words_to_now'
import times from 'lodash/times'

const StarRating = ({ rating }) => (
```

```

    <div>
      {times(rating, i => (
        <Star key={i} />
      ))}
      {times(5 - rating, i => (
        <StarBorder key={i} />
      ))}
    </div>
  )
}

class Review extends Component {
  state = {
    anchorEl: null
  }

  openMenu = event => {
    this.setState({ anchorEl: event.currentTarget })
  }

  closeMenu = () => {
    this.setState({ anchorEl: null })
  }

  edit = () => {
    this.closeMenu()
  }

  delete = () => {
    this.closeMenu()
  }

  toggleFavorite = () => {}

  render() {
    const {
      review: { text, stars, createdAt, favorited, author }
    } = this.props

    const linkToProfile = child => (
      <a
        href={`https://github.com/${author.username}`}
        target="_blank"
        rel="noopener noreferrer"
      >
        {child}
      </a>
    )

    return (
      <div>
        <Card className="Review">
          <CardHeader
            avatar={linkToProfile(
              <Avatar alt={author.name} src={author.photo} />
            )}
            action={
              <IconButton onClick={this.openMenu}>
                <MoreVert />
              </IconButton>
            }
            title={linkToProfile(author.name)}
            subheader={stars && <StarRating rating={stars} />}
          />
          <CardContent>
            <Typography component="p">{text}</Typography>
          </CardContent>
          <CardActions>
            <Typography className="Review-created">
              {distanceInWordsToNow(createdAt)} ago
            </Typography>
            <div className="Review-spacer" />
            <IconButton onClick={this.toggleFavorite}>
              {favorited ? <Favorite /> : <FavoriteBorder />}
            </IconButton>
          </CardActions>
        </Card>
      </div>
    )
  }
}

```

```

    </CardActions>
  </Card>
  <Menu
    anchorEl={this.state.anchorEl}
    open={Boolean(this.state.anchorEl)}
    onClose={this.closeMenu}
  >
    <MenuItem onClick={this.edit}>Edit</MenuItem>
    <MenuItem onClick={this.delete}>Delete</MenuItem>
  </Menu>
</div>
)
}
}

Review.propTypes = {
  review: PropTypes.shape({
    id: PropTypes.string.isRequired,
    text: PropTypes.string.isRequired,
    stars: PropTypes.number,
    createdAt: PropTypes.number.isRequired,
    favorited: PropTypes.bool,
    author: PropTypes.shape({
      name: PropTypes.string.isRequired,
      photo: PropTypes.string.isRequired,
      username: PropTypes.string.isRequired
    })
  }).isRequired
}

export default Review

```

The `MoreVert` button controls whether the `Menu` is open and where it is placed (or "anchored"). Also, in the `propTypes`, we list out all the fields of `review` that we use in `<Review>`.

We should now see a list of the 20 most recent reviews! 🐶

## Optimistic updates

If you're jumping in here, `git checkout 10_0.2.0` (tag `10_0.2.0`, or compare `10...11`)

Optimistic UI is when the client acts as if a user action has immediate effect instead of waiting for a response from the server. For example, normally if the user adds a comment to a blog post, the client sends the mutation to the server, and when the server responds with the new comment, the client adds it to the store, which updates the comment query results, which re-renders the page. Optimistic UI is when the client sends the mutation to the server and updates the store at the same time, not waiting for a response—*optimistically* assuming that the comment will be successfully saved to the database.

Let's write a simple example of an optimistic update for favoriting or unfavoriting a review. We can find in the [Playground](#) a mutation called `favoriteReview` which takes the review ID and whether the user is favoriting or unfavoriting. First we write the mutation and wrap `<Review>` with it:

`src/components/Review.js`

```

import gql from 'graphql-tag'
import { graphql } from 'react-apollo'

Review.propTypes = {
  review: ...
  favorite: PropTypes.func.isRequired
}

const FAVORITE_REVIEW_MUTATION = gql`
  mutation FavoriteReview($id: ObjectId!, $favorite: Boolean!) {
    favoriteReview(id: $id, favorite: $favorite) {

```



```

      favorited
    }
  }
}

export default graphql(FAVORITE_REVIEW_MUTATION, { name: 'favorite' })(Review)

```

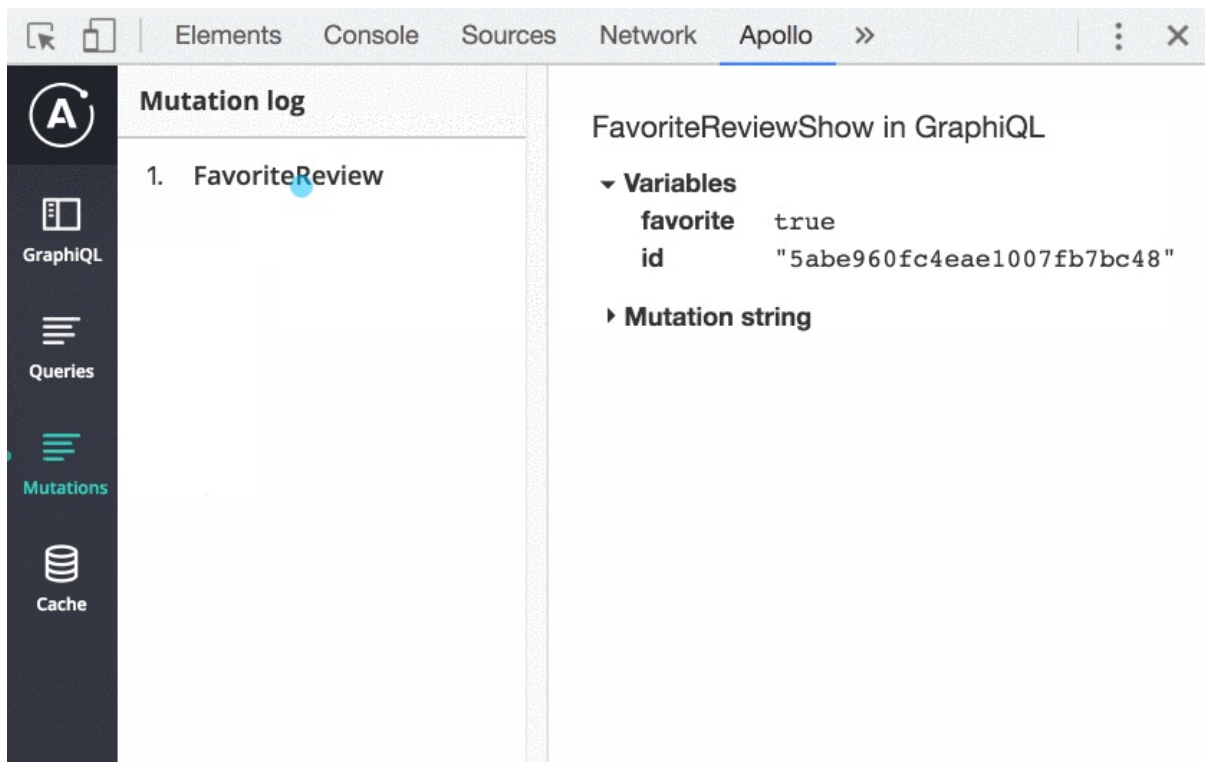
Then we have access to a `favorite` prop, which we use in the button click handler:

```

toggleFavorite = () => {
  const { review: { id, favorited } } = this.props
  this.props.favorite({
    variables: {
      id,
      favorite: !favorited
    }
  })
}

```

Now when we click a review's heart outline icon, it should change to the filled-in icon... right? 😊 But nothing's happening. Let's investigate with [Apollo devtools](#). We can open it on our page to the Mutations section. Then when we click a favorite button, `FavoriteReview` shows up in the Mutation log. So we know the mutation is getting called. And when we click on the log entry, we can see that the argument variables are given correctly:



So maybe the issue is with the server's response? Let's look at that in the Network tab. In the Name section on the left, scroll down to the bottom, and when we click the favorite button again, a new entry should appear. When we click on that, we should see the Headers tab, which at the top says it was an HTTP POST to

`https://api.graphql.guide/graphql` (which is the case for all of our GraphQL queries and mutations). It also says the response status code was "200 OK", so we know the server responded without an error. If we scroll to the bottom, we'll see the Request Payload, which has `operationName: FavoriteReview` and the correct mutation string and variables. Now if we switch to the Response tab, we see:

```

{"data":{"favoriteReview":{"favorited":true,"__typename":"Review"}}}

```

The server is giving us the correct response, so it looks like the mutation did succeed. Let's try reloading the page. Now we see that the review did get favorited. Why was the UI not updating? We forgot to include `id` in the response selection set, so Apollo didn't know which part of the store to update with `favorited: true`. When we add `id`, it works:

```
const FAVORITE_REVIEW_MUTATION = gql`
  mutation FavoriteReview($id: ObjID!, $favorite: Boolean!) {
    favoriteReview(id: $id, favorite: $favorite) {
      id
      favorited
    }
  }
`
```

#### *gif: Delayed favoriting*

While it works now, we can probably notice a delay between when we click the heart and when it changes. If we don't, we can switch from "Online" to "Fast 3G" in the dropdown on the far right top of the Network tab in Chrome devtools (which simulates the higher latency of mobile networks), and we'll notice a two-second delay before the icon changes. Users of our app who are on mobile or on computers far away from our servers notice the delay. Let's improve their experience by updating the icon immediately. (In reality, it will take some milliseconds to run the Apollo and React code and paint a new screen, but the delay should be imperceptible.)

We can provide an `optimisticResponse` to our `favorite()` mutation:

`src/components/Review.js`

```
toggleFavorite = () => {
  const { review: { id, favorited } } = this.props
  this.props.favorite({
    variables: {
      id,
      favorite: !favorited
    },
    optimisticResponse: {
      favoriteReview: {
        __typename: 'Review',
        id,
        favorited: !favorited
      }
    }
  })
}
```

`__typename` is an automatically provided field for the type being returned. We're mimicking the response from the server, which we saw had `"__typename": "Review"`:

```
{"data":{"favoriteReview":{"favorited":true,"__typename":"Review"}}
```

The type name, along with the `id`, will allow Apollo to figure out which review object in the store to update with the new `favorited` value. Now we see that the icon updates right away, even when we set the network speed to fast or slow 3G.

#### *gif: Optimistic favoriting*

We may find it helpful to decouple our presentational components from our data-fetching logic. Right now `<Review>` needs to know how to construct an `optimisticResponse` in order to call a mutation. We can make the separation cleaner by taking care of it outside the component:

`src/components/Review.js`

```
const withFavoriteMutation = graphql(FAVORITE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
```

```

    favorite: (id, favorite) =>
      mutate({
        variables: { id, favorite },
        optimisticResponse: {
          favoriteReview: {
            __typename: 'Review',
            id,
            favorited: favorite
          }
        }
      })
    })
  })
})

export default withFavoriteMutation(Review)

```

As we did with our queries, we can use `props` to control what props are given to `<Review>`. Here we're giving a `favorite` function that just takes the two pieces of data we need from the component, and then calls the mutation with the `variables` and `optimisticResponse` objects. Now we can simplify `toggleFavorite`:

```

class Review extends Component {

  toggleFavorite = () => {
    const { review: { id, favorited } } = this.props
    this.props.favorite(id, !favorited)
  }

  ...
}

```

`Review` no longer needs to know a special argument format for a mutation—it just gets a simple `favorite()` function to call.

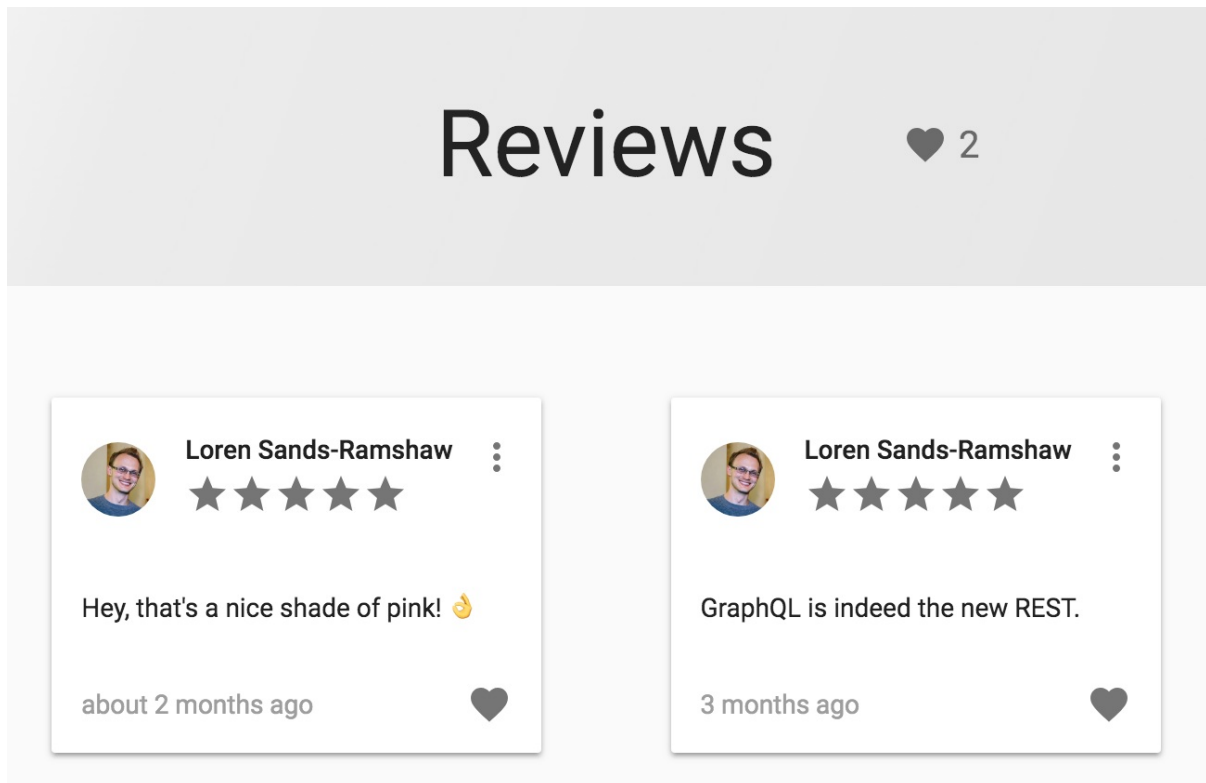
In the next section, we'll implement a more flexible and complex form of optimistic updating.

## Arbitrary updates

If you're jumping in here, `git checkout 11_0.2.0` (tag `11_0.2.0`, or compare `11...12`)

In the previous section ([Optimistic updating](#)), we changed the Apollo data store using `mutate()`'s `optimisticResponse` option. But that method only let us set the mutation response—an object of type `Review`. Sometimes we need to update different parts of the store. For our next piece of UI, we'll need to update the `User` object, and we'll do so with some new functions—`store.readQuery()` and `store.writeQuery()`.

In the header of the Reviews page, let's add the total count of favorited reviews:



First we need to think about how to get the count. We can't just count how many reviews in the store have `favorited: true`, because we only have the most recent 20. And fetching all the reviews from the server would be a lot of data on the wire, a lot of memory taken up on the client, and a long list to count through. Instead let's fetch the current user's `favoriteReviews` field. When we want to know more about the current user, we need to go back to our `withUser()` HOC and add the field to our `USER_QUERY`:

`src/lib/withUser.js`

```
const USER_QUERY = gql`
  query UserQuery {
    currentUser {
      ...
      favoriteReviews {
        id
      }
    }
  }
`
```

Since we're just counting the length, we don't need many `Review` fields—just the `id`. We get the data to `<Reviews>` using `withUser()`, and then we get the length of the `user.favoriteReviews` array to display on the page:

`src/components/Reviews.js`

```
import { graphql, compose } from 'react-apollo'
import get from 'lodash/get'
import { Favorite } from '@material-ui/icons'

import { withUser } from '../lib/withUser'

const Reviews = ({ reviews, loading, user }) => {
  const favoriteCount = get(user, 'favoriteReviews.length')

  return (
    <main className="Reviews mui-fixed">
      <div className="Reviews-header-wrapper">
        <header className="Reviews-header">
          {favoriteCount ? (
```

```

        <div className="Reviews-favorite-count">
          <Favorite />
          {favoriteCount}
        </div>
      ) : null}
    <h1>Reviews</h1>
  </header>
  ...
)
}

Reviews.propTypes = {
  ...
  user: PropTypes.shape({
    favoriteReviews: PropTypes.arrayOf(
      PropTypes.shape({
        id: PropTypes.string.isRequired
      })
    )
  })
}

export default compose(
  withReviews,
  withUser
)(Reviews)

```

Now if we have a non-zero favorite count, we should see it in the Reviews header. When we favorite reviews, the count doesn't go up as it should. We have to reload the page in order to get the count displayed accurately again—the user's `favoriteReviews` list is getting updated on the server, but not on the client. In order to update it on the client, we add another option to our mutation: `update`.

```

import remove from 'lodash/remove'

const READ_USER_FAVORITES = gql`
  query ReadUserFavorites {
    currentUser {
      id
      favoriteReviews {
        id
      }
    }
  }
`

const withFavoriteMutation = graphql(FAVORITE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
    favorite: (id, favorite) =>
      mutate({
        variables: { id, favorite },
        optimisticResponse: {
          favoriteReview: {
            __typename: 'Review',
            id,
            favorited: favorite
          }
        },
      }),
    update: store => {
      const data = store.readQuery({ query: READ_USER_FAVORITES })

      if (favorite) {
        data.currentUser.favoriteReviews.push({ id, __typename: 'Review' })
      } else {
        remove(data.currentUser.favoriteReviews, { id })
      }

      store.writeQuery({ query: READ_USER_FAVORITES, data })
    }
  })
})

```

`update` is given a `DataProxy` object, which allows us to read and write data from and to the store. To read data, we write a query for the data we want to change (in this case `currentUser.favoriteReviews`). To differentiate between queries we send to the server and queries we write just for reading from the store, we start the name with "Read": `ReadUserFavorites`. We give the query to `store.readQuery()`, and we get back the data. Then we modify the data (either adding or removing a bare-bones `Review` object with an `id` and `__typename`). Finally, we write the modified data back to the store with `store.writeQuery()`.

For example, if we started out with:

```
data = {
  currentUser: {
    __typename: 'User',
    favoriteReviews: [{
      __typename: 'Review',
      id: 'foo'
    }]
  }
}
```

and we favorited a review with ID `'bar'`, then we would write this data object back to the store:

```
{
  currentUser: {
    __typename: 'User',
    favoriteReviews: [{
      __typename: 'Review',
      id: 'foo'
    }, {
      __typename: 'Review',
      id: 'bar'
    }]
  }
}
```

Then Apollo would update `USER_QUERY`'s `user` prop, which would be passed down to `<Reviews>`, which would find a new `user.favoriteReviews.length` value and re-render the component. We can see that this process works in our app:

*gif: Updating favorite count*

In the [next section](#), we'll write an `update()` function that adds an item to a list. We can also use `readQuery()` and `writeQuery()` outside of a mutation—we can wrap any component in `withApollo()`, and then inside the component call, for instance: `this.props.client.writeQuery()`.

There are two more functions we can use—`readFragment()` and `writeFragment()`. `readQuery` can only read data from a root query type like `currentUser{ ... }` or `reviews(limit: 20){ ... }`. `readFragment` can read from any normalized object in our store by its store ID.

A *store ID* is the identifier Apollo uses to [normalize](#) objects. By default, it is `['__typename']: [id]`, for instance: `Review: 5a6676ec094bf236e215f488`. We can see these IDs on the left of the Store section in Apollo devtools:

The screenshot shows the Apollo Client DevTools interface. On the left, the 'Cache' tab is selected, showing a list of store IDs. The main area displays the JSON representation of the cache state.

**Cache Tab (Left Sidebar):**

- Search...
- ROOT\_QUERY
- Chapter:-1
- Chapter:-2
- Chapter:-3
- Chapter:0
- Chapter:1
- Chapter:5
- ROOT\_MUTATION
- Review:5abe941d25aca9
- Review:5abe960fc4eae1
- Section:1-2
- Section:1-intro
- Section:5-1
- Section:5-2
- Section:foreword
- Section:intro

**Cache State (Main Area):**

```

▼ ROOT_QUERY
  chapters: [Chapter]
    0: ▶ Chapter:-3
    1: ▶ Chapter:-2
    2: ▶ Chapter:-1
    3: ▶ Chapter:0
    4: ▶ Chapter:1
    5: ▶ Chapter:5
  currentUser: User
    email: "lorensr@gmail.com"
    favoriteReviews:
    firstName: "Loren"
    hasPurchased: "FULL_COURSE"
    name: "Loren Sands-Ramshaw"
    photo: "https://avatars2.githubusercontent.com/u/251288?v=4"
    username: "lorensr"
  githubStars: 10
  reviews({"limit":10,"orderBy":"createdAt_DESC"}): [Review]
    0: ▶ Review:5abe960fc4eae1007fb7bc48
    1: ▶ Review:5abe941d25aca9fe2306cff9
  section({"id":"intro"}): Section
    ▶ Section:intro
  
```

On the left is the store IDs of all objects in the store. There are reviews with their random IDs, as well as sections with store IDs like `Section:1-1`. We can read a section by its store ID like this:

```

this.props.client.readFragment({
  id: 'Section:introduction',
  fragment: gql`
    fragment exampleSection on Section {
      id
      views
      content
    }
  `,
})

```

The `readFragment()` arguments are the store ID and a [fragment](#). It returns just that section:

```

{
  content: "...",
  id: "introduction",
  views: 67,
  __typename: "Section",
  Symbol(id): "Section:intro"
}

```

Similarly, `writeFragment()` allows us to write to an object with a specific store ID:

```

this.props.client.writeFragment({
  id: 'Section:intro',
  fragment: gql`
    fragment sectionContent on Section {
      content
      __typename
    }
  `,
  data: {
    content: 'overwritten',
    __typename: 'Section'
  }
})

```

If we ran this and then navigated to `/Introduction`, the section text would have changed to just the word "overwritten" 😄. Not to worry—it's just changing the local client-side store; when we reload, the actual Introduction text gets refetched from the server. We can try it out in the console, but first we have to (temporarily) add this line in any of our js files that imports `gql`:

```
window.gql = gql
```

And then we replace `this.props.client` with `__APOLLO_CLIENT__`, which is a global variable available in development.

*gif: Writing a fragment to the store*

## Creating reviews

If you're jumping in here, `git checkout 12_0.2.0` (tag `12_0.2.0`, or compare `12...13`)

Adding the ability to create reviews will give us the opportunity to look at a more complex mutation and a different kind of `update()` function—we'll be updating our list of reviews with a new review so that it shows up at the top of the Reviews page.

Let's start out by adding a FAB (floating action button) that appears on the Reviews page when the user is logged in. The FAB will open a modal that has the form for a new review. Whether the modal is open is a state variable, so we need to convert `<Reviews>` from a function to a stateful component:

`src/components/Reviews.js`

```
import React, { Component } from 'react'
import { Fab, Modal } from '@material-ui/core'
import { Add } from '@material-ui/icons'

import AddReview from './AddReview'

class Reviews extends Component {
  state = {
    addingReview: false
  }

  addReview = () => {
    this.setState({ addingReview: true })
  }

  doneAddingReview = () => {
    this.setState({ addingReview: false })
  }

  render() {
    const { reviews, loading, user } = this.props

    ...

    {user && (
      <div>
        <Fab
          onClick={this.addReview}
          color="primary"
          className="Reviews-add"
        >
          <Add />
        </Fab>

        <Modal
          open={this.state.addingReview}
          onClose={this.doneAddingReview}
        >
          <AddReview done={this.doneAddingReview} user={user} />
        </Modal>
      </div>
    )}
```



```

    </div>
  )
}
}
}

```

`<AddReview>` will need a way to let us know it's done (so we can close the modal) and will need to know who the user is (the creator of the review). To set a primary color for the FAB that matches the rest of the site, we need a Material UI [theme](#). We can see from the [default theme](#) that `palette.primary.main` is the name of the value to change:

`src/index.js`

```

import { MuiThemeProvider, createMuiTheme } from 'material-ui/styles'

const GRAPHQL_PINK = '#e10098'

const theme = createMuiTheme({
  palette: { primary: { main: GRAPHQL_PINK } }
})

ReactDOM.render(
  <BrowserRouter>
    <ApolloProvider client={client}>
      <MuiThemeProvider theme={theme}>
        <App />
      </MuiThemeProvider>
    </ApolloProvider>
  </BrowserRouter>,
  document.getElementById('root')
)

```

Next up is the `<AddReview>` form:

`src/components/AddReview.js`

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'
import StarInput from 'react-star-rating-component'
import { Button, TextField } from '@material-ui/core'
import { Star, StarBorder } from '@material-ui/icons'

import { validateReview } from '../lib/validators'

const GREY = "#0000008a"

class AddReview extends Component {
  state = {
    text: '',
    stars: null,
    errorText: null
  }

  updateText = event => {
    this.setState({ text: event.target.value })
  }

  updateStars = stars => {
    this.setState({ stars })
  }

  handleSubmit = event => {
    event.preventDefault()
    const { text, stars } = this.state

    const errors = validateReview({ text, stars })
    if (errors.text) {
      this.setState({ errorText: errors.text })
      return
    }
  }
}

```

```

    }

    // mutate

    this.props.done()
  }

  render() {
    return (
      <form
        className="AddReview"
        autoComplete="off"
        onSubmit={this.handleSubmit}
      >
        <TextField
          className="AddReview-text"
          label="Review text"
          value={this.state.text}
          onChange={this.updateText}
          helperText={this.state.errorText}
          error={!this.state.errorText}
          multiline
          rowsMax="10"
          margin="normal"
          autoFocus={true}
        />

        <StarInput
          className="AddReview-stars"
          starCount={5}
          editing={true}
          value={this.state.stars}
          onStarClick={this.updateStars}
          renderStarIcon={({currentStar, rating}) =>
            currentStar > rating ? <StarBorder /> : <Star />
          }
          starColor={GREY}
          emptyStarColor={GREY}
          name="stars"
        />

        <div className="AddReview-actions">
          <Button className="AddReview-cancel" onClick={this.props.done}>
            Cancel
          </Button>

          <Button type="submit" color="primary" className="AddReview-submit">
            Add review
          </Button>
        </div>
      </form>
    )
  }
}

AddReview.propTypes = {
  done: PropTypes.func.isRequired,
  user: PropTypes.shape({
    name: PropTypes.string.isRequired,
    photo: PropTypes.string.isRequired,
    username: PropTypes.string.isRequired
  }).isRequired
}

export default AddReview

```

Before we mutate, we need to validate the form input and show the error message, if any. We'll use the [revalidate](#) library:

```
src/lib/validators.js
```

```
import {
```

```

    createValidator,
    composeValidators,
    combineValidators,
    isRequired,
    hasLengthLessThan
  } from 'revalidate'

  const isString = createValidator(
    message => value => {
      if (!(typeof value === 'string')) {
        return message
      }
    },
    field => `${field} must be a String`
  )

  export const validateReview = combineValidators({
    text: composeValidators(isRequired, isString, hasLengthLessThan(500))(
      'Review text'
    ),
    stars: createValidator(
      message => value => {
        if (![null, 1, 2, 3, 4, 5].includes(value)) {
          return message
        }
      },
      field => `${field} must be a number 1-5`
    )('Stars')
  })

```

We use `createValidator` to create custom validator functions, `composeValidator` to compose multiple validator functions together, and `combineValidators` to combine our validators in an object matching our data format, with `text` and `stars` fields. Here are some example outputs:

```

validateReview({
  text: 1,
  stars: 5
})

// => {text: "Review text must be a String"}

validateReview({
  text: 'my review',
  stars: 'a string'
})

// => {stars: Stars must be a number 1-5}

```

We don't need to check for a `stars` error because our `<StarInput>` doesn't produce an invalid value. But we include it in the validator because we'll also use it on the server.

Next we add the mutation! In the [Playground](#) we find the `createReview` mutation. (The convention is that if the data type is `Foo`, the basic **CUD** mutations are called `createFoo`, `updateFoo`, and `deleteFoo`.) We're used to `gql` and the `graphql` and `props` functions, but this time we'll have a larger `optimisticResponse` and a different kind of `update()`:

`src/components/AddReview.js`

```

import gql from 'graphql-tag'
import { graphql } from 'react-apollo'

class AddReview extends Component {
  ...

  handleSubmit = event => {
    event.preventDefault()
    const { text, stars } = this.state

```

```

const errors = validateReview({ text, stars })
if (errors.text) {
  this.setState({ errorText: errors.text })
  return
}

this.props.addReview(text, stars)

this.props.done()
}

...
}

AddReview.propTypes = {
  ...
  addReview: PropTypes.func.isRequired
}

const ADD_REVIEW_MUTATION = gql`
  mutation AddReview($input: CreateReviewInput!) {
    createReview(input: $input) {
      id
      text
      stars
      createdAt
      favorited
      author {
        name
        photo
        username
      }
    }
  }
`

const withMutation = graphql(ADD_REVIEW_MUTATION, {
  props: ({ ownProps: { user }, mutate }) => ({
    addReview: (text, stars) => {
      mutate({
        variables: {
          input: { text, stars }
        },
      },
      optimisticResponse: {
        createReview: {
          __typename: 'Review',
          id: null,
          text,
          stars,
          createdAt: new Date(),
          favorited: false,
          author: {
            __typename: 'User',
            id: null,
            name: user.name,
            photo: user.photo,
            username: user.username
          }
        }
      }
    ),
    update: (store, { data: { createReview: newReview } }) => {
      const data = store.readQuery({
        query: TODO
      })
      data.reviews.unshift(newReview)
      store.writeQuery({ query: TODO, data })
    }
  })
})
}
})
})
})

export default withMutation(AddReview)

```

We don't know what the server-side `id` will be, so we set it to `null`, and it will be updated by Apollo when the server response arrives. Similarly, `createdAt` will be a little different on the server, but not enough to make a difference for optimistic display. We know that `favorited` is `false` because the user hasn't had a chance to favorite the new review, and the `author` is the current user.

So far our mutations have updated an existing object in the store (the one with the same `id`), and that object, since it was part of a query result, triggers a component re-render. But this time there is no existing object: we're adding a new object to the store. And the new object isn't part of a query result. Apollo will add an object of type `Review` with `id: null` to the store, but it won't update the `<Reviews>` component's `reviews` prop because Apollo doesn't know the new review object should be part of the `REVIEWS_QUERY` results. So we have to change the `REVIEWS_QUERY` results ourselves in the `update` function.

But first we need access to `REVIEWS_QUERY`, a variable inside `Reviews.js`. We'd run into trouble setting it as a static property on `Reviews` and doing `import Reviews from './Reviews'` because we'd have an import cycle—`Reviews.js` imports `AddReview` (`Reviews` would wind up being `null`). So let's create a new folder for GraphQL documents, `src/graphql/`, and make a new file:

`src/graphql/Review.js`

```
import gql from 'graphql-tag'

export const REVIEWS_QUERY = gql`
  query ReviewsQuery {
    reviews(limit: 20) {
      id
      text
      stars
      createdAt
      favorited
      author {
        id
        name
        photo
        username
      }
    }
  }
`
```

And in `Reviews.js` and `AddReview.js`, we import it:

`src/components/AddReview.js`

```
import { REVIEWS_QUERY } from '../graphql/Review'

...

update: (store, { data: { createReview: newReview } }) => {
  const data = store.readQuery({
    query: REVIEWS_QUERY
  })
  data.reviews.unshift(newReview)
  store.writeQuery({ query: REVIEWS_QUERY, data })
}
```

The second parameter to `update` has the mutation response—it's called first with the optimistic response, and then with the server response. So initially, `data.createReview` is the `optimisticResponse.createReview` object we just created. First we call `readQuery`, reading the current results from the store. Then we modify the data, `unshift` ing the `newReview` onto the beginning of the array, so that it shows up first, at the top of the page.

*gif: Optimistically adding review*

## Using fragments

**Fragments** are good for more than just [reading from and writing to the store](#): they also can **DRY** up our queries and mutations. The selection set on `reviews` in the query we just relocated was the same as the selection set on `createReview` we used in our mutation. Let's put that selection set in a fragment:

`src/graphql/Review.js`

```
import gql from 'graphql-tag'

export const REVIEW_ENTRY = gql`
  fragment ReviewEntry on Review {
    id
    text
    stars
    createdAt
    favorited
    author {
      id
      name
      photo
      username
    }
  }
`

export const REVIEWS_QUERY = gql`
  query ReviewsQuery {
    reviews(limit: 20) {
      ...ReviewEntry
    }
  }
  ${REVIEW_ENTRY}
`
```

We can't name the fragment `Review` because that's a type name, so the convention is `ReviewEntry`. We can greatly simplify our `Review.propTypes` with the `propType()` function from [graphql-anywhere](#):

`src/components/Review.js`

```
import { propType } from 'graphql-anywhere'
import { REVIEWS_QUERY, REVIEW_ENTRY } from '../graphql/Review'

Review.propTypes = {
  review: propType(REVIEW_ENTRY).isRequired,
  favorite: PropTypes.func.isRequired
}
```

`propType()` generates a React `propTypes`-compatible type-checking function for the `review` object from our `ReviewEntry` fragment.

Let's also use the fragment in `<Reviews>` and `<AddReview>`:

`src/components/Reviews.js`

```
import { propType } from 'graphql-anywhere'
import { REVIEWS_QUERY, REVIEW_ENTRY } from '../graphql/Review'

Reviews.propTypes = {
  reviews: PropTypes.arrayOf(propType(REVIEW_ENTRY)),
}
```

`src/components/AddReview.js`

```
import { propType } from 'graphql-anywhere'
import { REVIEWS_QUERY, REVIEW_ENTRY } from '../graphql/Review'
```

```
const ADD_REVIEW_MUTATION = gql`
  mutation AddReview($input: CreateReviewInput!) {
    createReview(input: $input) {
      ...ReviewEntry
    }
  }
`
${REVIEW_ENTRY}
```

## Deleting

If you're jumping in here, `git checkout 13_0.2.0` (tag `13_0.2.0`, or compare `13...14`)

Next let's see how deleting an item works. We can add a dialog box confirming deletion, and when it's confirmed, we'll send the `removeReview(id)` mutation:

`src/components/Review.js`

```
import { graphql, compose } from 'react-apollo'
import Dialog, {
  DialogActions,
  DialogContent,
  DialogContentText,
  DialogTitle
} from 'material-ui/Dialog'
import Button from 'material-ui/Button'

class Review extends Component {
  state = {
    anchorEl: null,
    deleteConfirmationOpen: false
  }

  openDeleteConfirmation = () => {
    this.closeMenu()
    this.setState({ deleteConfirmationOpen: true })
  }

  closeDeleteConfirmation = () => {
    this.setState({ deleteConfirmationOpen: false })
  }

  delete = () => {
    this.closeDeleteConfirmation()
    this.props.delete(this.props.review.id)
  }

  ...

  <Dialog
    open={this.state.deleteConfirmationOpen}
    onClose={this.closeDeleteConfirmation}
  >
    <DialogTitle>{'Delete review?'}</DialogTitle>
    <DialogContent>
      <DialogContentText>
        A better UX is probably just letting you single-click delete with
        an undo toast, but that's harder to code right{' '}
        <span role="img" aria-label="grinning face">
          
        </span>
      </DialogContentText>
    </DialogContent>
    <DialogActions>
      <Button onClick={this.closeDeleteConfirmation}>Cancel</Button>
      <Button onClick={this.delete} color="primary" autoFocus>
        Sudo delete
      </Button>
    </DialogActions>
  </Dialog>
```

```

    </DialogActions>
  </Dialog>

  ...

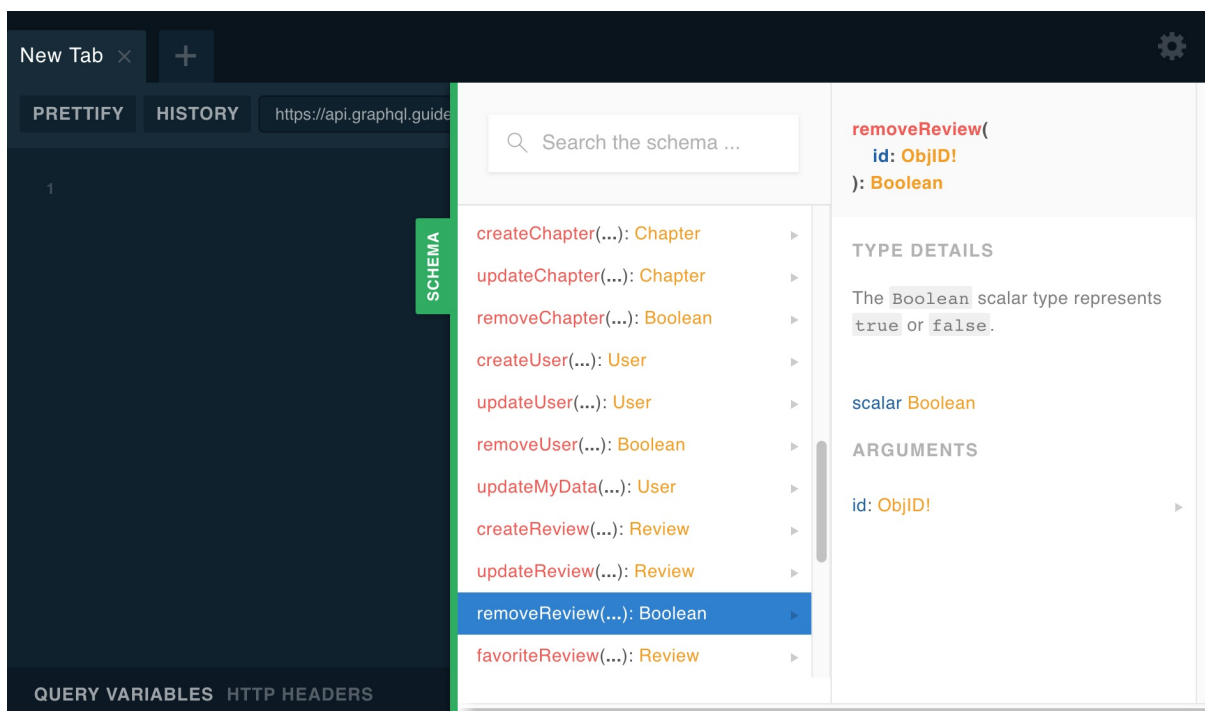
const DELETE_REVIEW_MUTATION = gql`
  mutation DeleteReview($id: ObjID!) {
    removeReview(id: $id)
  }
`

const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({ delete: id => mutate({ variables: { id } }) })
})

export default compose(withFavoriteMutation, withDeleteMutation)(Review)

```

We see in the [Playground schema](#) that `removeReview` resolves to a scalar type (`Boolean`), so unlike our previous mutations, it doesn't have a selection set:



When we try out the new delete dialog, we notice that the review remains on the page. Did it work? We can check on the devtools Network tab, selecting the last `graphql` request, and switching to the Response tab:

```

{"data":{"removeReview":true}}

```

*gif: Server response to removeReview*

So the deletion was successful (when we refresh the page, the review is gone), but Apollo client didn't know it should remove the review object from the store. We can tell it to do so with `update()`:

```

const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
    delete: id =>
      mutate({
        variables: { id },
        update: store => {
          let data = store.readQuery({ query: Review.queries.REVIEWS })
          remove(data.reviews, { id })
          store.writeQuery({ query: Review.queries.REVIEWS, data })
        }
      })
  })
})

```



```

    data = store.readQuery({ query: READ_USER_FAVORITES })
    remove(data.currentUser.favoriteReviews, { id })
    store.writeQuery({ query: READ_USER_FAVORITES, data })
  }
})
})
})
})

```

We need to remove the review not only from the `REVIEWS` query, but also from `currentUser.favoriteReviews` — otherwise, when we delete a favorited review, the count in the header of the reviews page will be inaccurate.

We're using `update()` without an `optimisticResponse`, which means it will only be called once, when the server response arrives. We'll notice a delay between clicking `SUDO DELETE` and the review being removed from the page. If we want it to be removed immediately, we need an `optimisticResponse`, even if we're not using the optimistic data:

```

mutate({
  variables: { id },
  optimisticResponse: {
    removeReview: true
  },
  update: ...

```

*gif: Removing a review*

## Error handling

Background: [GraphQL errors](#)

If you're jumping in here, `git checkout 14_0.2.0` (tag `14_0.2.0`, or compare `14...15`)

When we try to delete a review that isn't ours, nothing happens. In the console, we see:

```

ApolloError.js:34 Uncaught (in promise) Error: GraphQL error: unauthorized
    at new ApolloError (ApolloError.js:34)
    at Object.next (QueryManager.js:98)
    at SubscriptionObserver.next (zen-observable.js:154)
    at SubscriptionObserver.next (zen-observable.js:154)
    at httpLink.js:140
    at <anonymous>

```

Let's break that down:

- `ApolloError.js:34 Uncaught (in promise) Error:` —Apollo is saying that there was a Promise that threw an error, and our code didn't catch it.
- `GraphQL error:` —It was a GraphQL error: an error returned to us from the GraphQL server, not an error in the Apollo library.
- `unauthorized` —This is the error message from the GraphQL server

So the Guide server is saying that we're not authorized to execute that `removeReview` mutation. This makes sense, because it's not our review. We should have the app tell the user that, though. A call to `mutate()` —or, in our case, `this.props.delete` —returns a Promise. This Promise will throw GraphQL errors, which we can catch like this:

```

this.props
  .delete(this.props.review.id)
  .catch(e => console.log(e.graphQLErrors))

```

`e.graphQLErrors` is an array of all the errors returned from the server. In this case, we just have one:

```
[
  {
    message: "unauthorized",
    locations: [{ "line": 2, "column": 3 }],
    path: ["removeReview"]
  }
]
```

We can now alert the user of the error, depending on whether we find an "unauthorized" message:

src/components/Review.js

```
delete = () => {
  this.closeDeleteConfirmation()
  this.props.delete(this.props.review.id).catch(e => {
    if (find(e.graphQLErrors, { message: 'unauthorized' })) {
      alert(' ❗ You can only delete your own reviews!')
    }
  })
}
```

But what about other errors? We could get errors about anything bad happening on the server, from dividing by zero to a database query failing. We could add an `else` statement:

```
} else {
  alert('Unexpected error occurred')
}
```

But that wouldn't cover unexpected errors occurring in all of our other queries and mutations. We can avoid peppering these unexpected-error alerts all over our code by checking errors globally as they arrive from the network. Whenever we want to do some logic that all requests or responses go through, we use a link. At the end of the [Logging in](#) section, we used an `apollo-link-context` to set an authentication header on all outgoing HTTP requests. Here we can use an `apollo-link-error`. In `index.js`, we rename our `link` to be `networkLink`, then:

src/index.js

```
import { errorLink } from './lib/errorLink'

const link = errorLink.concat(networkLink)
```

In a chain of links from left to right (where `leftLink.concat(rightLink)`), off the left side of the chain is our code, and off the right side is the network. We put `errorLink` to the left of `networkLink` because we need the GraphQL response coming from the network (off right side) to first go through the `networkLink` (the right end), and then to the `errorLink` (left end), before reaching our code (off left side). We create a new file for `errorLink`:

src/lib/errorLink.js

```
import { onError } from 'apollo-link-error'

const KNOWN_ERRORS = ['unauthorized']

export const errorLink = onError(({ graphQLErrors, networkError }) => {
  if (networkError) {
    console.log(`[Network error]: ${networkError}`)
    return
  }

  if (graphQLErrors) {
    const unknownErrors = graphQLErrors.filter(
      error => !KNOWN_ERRORS.includes(error.message)
    )

    if (unknownErrors.length) {
```

```

    alert('    An unexpected error occurred on the server')
    unknownErrors.map(({ message, locations, path }) =>
      console.log(`[GraphQL error]: Message: ${message}, Path: ${path}`)
    )
  }
}
})

```

If there's a known error, like `'unauthorized'`, let's leave it to the originating component to alert the user, since that component knows the context of the error. For example, in `<Review>`, we can be specific, saying "You can only delete your own reviews!" Whereas if we made the alert in `errorLink`, it would be less helpful: "You are not authorized to view this data or perform this action."

By default, when a GraphQL error is returned from the server, Apollo treats it as a fatal error in the query or mutation. In the case of an unauthorized deletion, the error is thrown from the mutation function, and `update()` isn't called. This is why the review remains on the page. If we were sending a mutation for which we didn't care about server errors, and we wanted the `update()` function to always run regardless, we could change the mutation's default [error policy](#):

[src/components/Review.js](#)

```

const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
  options: { errorPolicy: 'ignore' },
  props: ({ mutate }) => ...
})

```

Then the call to `this.props.delete()` would resolve without error, and the review would be removed from the store and page.

Changing the error policy is more often useful when querying. Let's see how the default error policy works when querying. We can change the `limit` argument on our `reviews` query to a special value of `-1` that will return demo reviews, some of which have a private `text` field.

[src/graphql/Review.js](#)

```

export const REVIEWS_QUERY = gql`
  query ReviewsQuery {
    reviews(limit: -1) {

```

When we do this query in [Playground](#):

```

{
  reviews(limit: -1) {
    stars
    text
  }
}

```

here's the response we get back:

```

{
  "data": {
    "reviews": [
      {
        "stars": 5,
        "text": null
      },
      {
        "stars": 4,
        "text": "GraphQL is awesome, but React is soooo 2016. Write me a Vue chapter!"
      },
      {
        "stars": 3,
        "text": null
      }
    ]
  }
}

```

```

    }
  ],
},
"errors": [
  {
    "message": "unauthorized",
    "locations": [
      {
        "line": 4,
        "column": 5
      }
    ],
    "path": [
      "reviews",
      0,
      "text"
    ]
  },
  {
    "message": "unauthorized",
    "locations": [
      {
        "line": 4,
        "column": 5
      }
    ],
    "path": [
      "reviews",
      2,
      "text"
    ]
  }
]
}
}
}

```

**Playground:** `query { reviews(limit: -1) { stars text } }`

The first and third reviews have private `text` fields, so we see `text: null` in `data.reviews` and the `errors` array has entries for each one with "unauthorized" messages. The first error path is `reviews.0.text`, corresponding to the 0th review in the `data.reviews` array, and the second error is at `review.2.text`. So the errors match up with the reviews that have `text: null`.

The Review schema says that `text` is nullable. If `text` had been non-nullable (`text: String!`), then an error in the `text` resolver would have made the entire object `null` — `data` would have been `{ "reviews": null }`.

Let's see how our app is handling this partially-null data response with an `errors` attribute. We're getting an error:

```

Uncaught TypeError: Cannot read property 'map' of undefined
    at Reviews.render (Reviews.js:46)
    ...

```

Which corresponds to this line:

```
reviews.map(review => <Review key={review.id} review={review} />)
```

So it looks like `reviews` is undefined. Let's also look at `data.error`:

`src/components/Reviews.js`

```

class Reviews extends Component
  render() {
    console.log(this.props.error)
    ...
  }
}

```

```
const withReviews = graphql(Review.queries.REVIEWS, {
  props: ({ data: { reviews, loading, error } }) => ({
    reviews,
    loading,
    error
  })
})
```

It has these fields:

```
["stack", "graphQLErrors", "networkError", "message", "extraInfo"]
```

and `this.props.error.graphQLErrors` looks like this:

```
[
  {
    message: "unauthorized",
    locations: [{ line: 10, column: 3 }],
    path: [ "reviews", 0, "text" ]
  },
  {
    message: "unauthorized",
    locations: [{ line: 10, column: 3 }],
    path: [ "reviews", 2, "text" ]
  }
]
```

If we want `reviews` to be defined, we can set `errorPolicy` to `'all'`:

```
src/components/Review.js
```

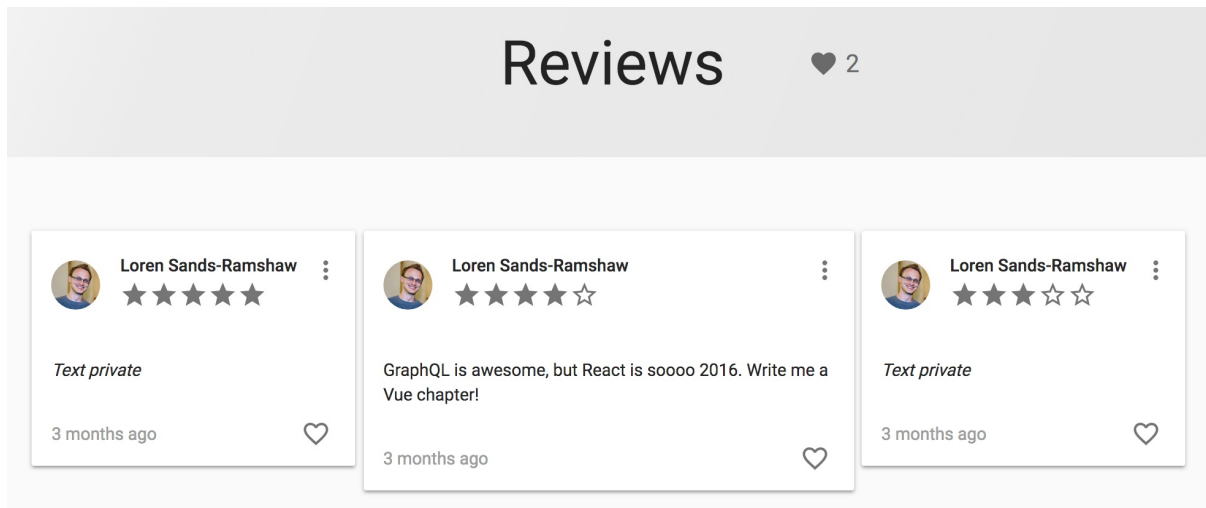
```
const withReviews = graphql(Review.queries.REVIEWS, {
  options: { errorPolicy: 'all' },
```

We can handle `text` sometimes being `null` in `<Review>`:

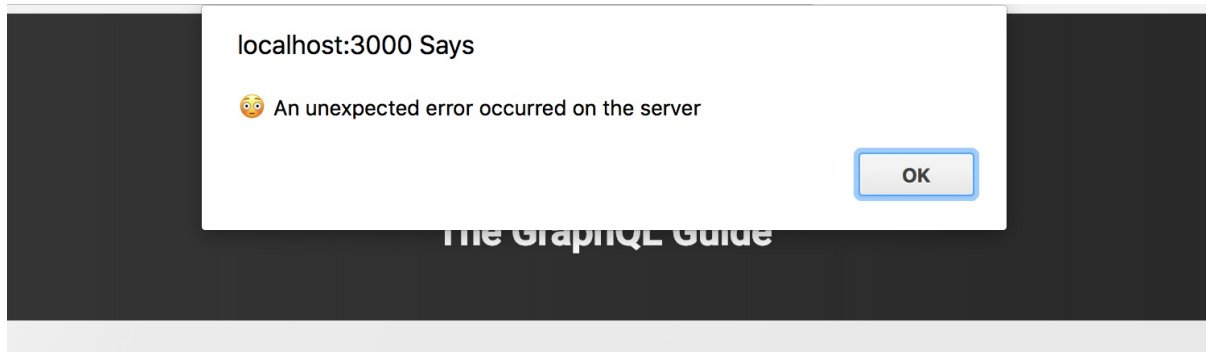
```
src/components/Review.js
```

```
<CardContent>
  {text ? (
    <Typography component="p">{text}</Typography>
  ) : (
    <Typography component="i">Text private</Typography>
  )}
</CardContent>
```

If there were other errors that we thought might result in a null `text` field, we could take different actions based on `this.props.error` in `<Reviews>`. If we wanted to ignore all errors (reviews would be defined, and `this.props.error` would be undefined), we could set `errorPolicy: 'ignore'`.



Let's see what happens when we trigger a different error: first let's sign out, and then let's interact with a review. We notice that when we favorite, edit, or delete, the "unexpected error" alert appears:



To figure out what it is, we could look at the GraphQL response in the Network panel, or we can just look in the console, since the `errorLink` we made logs unknown errors. There, we find that the error message is `must sign in`, for instance:

```
[GraphQL error]: Message: must sign in, Path: favoriteReview
```

Having a user see this alert isn't good UX. One way to avoid it is by adding `must sign in` to `KNOWN_ERRORS` in `src/lib/errorLink.js`, and then handling the error in `<Review>` with a message like, "Sign in to favorite a review." Another way to avoid the error is to just remove the UI controls when the user isn't signed in 😬. Let's go with the latter solution, but before we do, note what happens to the review on the page right after we take the action, before we dismiss the alert: when we favorite, the heart stays filled in; when we delete, the review disappears, and when we edit, the review changes. In each case, when we dismiss the alert, the review changes back to its previous state. This is a great demonstration of optimistic updates—Apollo applies the optimistic change, then it receives an error back from the server, which goes through our `errorLink`, which puts up an alert, which halts JS execution until it is dismissed. Once it's dismissed, Apollo is able to finish handling the response—it realizes that the mutation was unsuccessful, so it rolls back the optimistic update, restoring our store to its previous state, which triggers new props being provided to our components, which triggers React to re-render them.

To remove the UI elements, we need to first get the user info down to `<Review>`:

```
src/components/Reviews.js
```

```
reviews.map(review => (
  <Review key={review.id} review={review} user={user} />
))
```

And then check if `user` is defined:

`src/components/Review.js`

```
render() {
  const {
    review: { text, stars, createdAt, favorited, author },
    user
  } = this.props


  ...

  <CardHeader
    action={
      user && (
        <IconButton onClick={this.openMenu}>
          <MoreVert />
        </IconButton>
      )
    }
  >
    ...

    {user && (
      <IconButton onClick={this.toggleFavorite}>
        {favorited ? <Favorite /> : <FavoriteBorder />}
      </IconButton>
    )}
  </CardHeader>

  ...


  Review.propTypes = {
    ...
    user: PropTypes.object
  }
}
```

★ 13 stars

Sign in

The GraphQL Guide


Preface
Introduction
1. Understanding GraphQL through REST
Introduction
GraphQL as an alternative to a REST API
A simple REST API server
A simple GraphQL server
Querying a set of data
Filtering the data
Async data loading
Multiple types of data
Security & error handling
Tying this all together
2. Query Language

# Reviews


Loren Sands-Ramshaw
★★★★★

Hey, that's a nice shade of pink! 🍷

about 2 months ago


Loren Sands-Ramshaw
★★★★★

GraphQL is indeed the new REST.

3 months ago

## Editing reviews

If you're jumping in here, `git checkout 15_0.2.0` (tag [15\\_0.2.0](#), or compare [15...16](#))

The last piece of reviews we haven't implemented yet is editing! Let's see how much of our `<AddReview>` component we can reuse by renaming it to `<ReviewForm>` and deciding which mutation to call based on the props. We'll need to add a `<Modal>` with the form to `<Review>` and pass in the review object as a prop:

`src/components/Review.js`

```
import Modal from 'material-ui/Modal'

import ReviewForm from './ReviewForm'

class Review extends Component {
  state = {
    anchorEl: null,
    deleteConfirmationOpen: false,
    editing: false
  }

  edit = () => {
    this.closeMenu()
    this.setState({ editing: true })
  }

  doneEditing = () => {
    this.setState({ editing: false })
  }

  render() {
    ...
    <Modal open={this.state.editing} onClose={this.doneEditing}>
      <ReviewForm done={this.doneEditing} review={this.props.review} />
    </Modal>
  </div>
  )
}
}
```

The mutation takes the review's `id` and the new `text` and `stars` fields:

```
input UpdateReviewInput {
  text: String!
  stars: Int
}

type Mutation {
  updateReview(id: ObjID!, input: UpdateReviewInput!): Review
}
```

We know whether we're editing based on the presence of the `review` prop, and we also use it to set initial values for the `text` and `stars` inputs:

`src/components/ReviewForm.js`

```
import { graphql, compose } from 'react-apollo'
import classNames from 'classnames'

class ReviewForm extends Component {
  constructor(props) {
    super(props)

    const { review } = props

    this.isEditing = !!review

    this.state = {
      text: review ? review.text : '',
      stars: review ? review.stars : null,
    }
  }
}
```



```

      errorText: null
    }
  }
  ...

  handleSubmit = event => {
    event.preventDefault()
    const { text, stars } = this.state

    const errors = validateReview({ text, stars })
    if (errors.text) {
      this.setState({ errorText: errors.text })
      return
    }

    const { review } = this.props

    if (this.isEditing) {
      this.props.editReview(review.id, text, stars)
    } else {
      this.props.addReview(text, stars)
    }

    this.props.done()
  }

  render() {
    return (
      <form
        className={classNames('ReviewForm', { editing: this.isEditing })}

        ...

        <Button type="submit" color="primary" className="AddReview-submit">
          {this.isEditing ? 'Save' : 'Add review'}
        </Button>
      </div>
    </form>
  )
}
}

ReviewForm.propTypes = {
  ...
  editReview: PropTypes.func.isRequired,
  review: PropTypes.shape({
    id: PropTypes.string.isRequired,
    text: PropTypes.string,
    stars: PropTypes.number
  })
}
...

const EDIT_REVIEW_MUTATION = gql`
  mutation EditReview($id: ObjID!, $input: UpdateReviewInput!) {
    updateReview(id: $id, input: $input) {
      id
      text
      stars
    }
  }
`

const withEditReview = graphql(EDIT_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
    editReview: (id, text, stars) => {
      mutate({
        variables: {
          id,
          input: { text, stars }
        },
      },
      optimisticResponse: {

```

```

      updateReview: {
        __typename: 'Review',
        id,
        text,
        stars
      }
    }
  })
}
})
})
})

export default compose(withAddReview, withEditReview)(ReviewForm)

```

When editing an object, we only need to select the `id` and fields that are changing. When the response arrives (and when the `optimisticResponse` is handled), just those fields are updated in the store (the other fields like `author` and `favorited` will remain).

*gif: Editing a review*

## Advanced querying

Section contents:

- [Paginating](#)
  - [Offset-based](#)
    - [page](#)
    - [skip & limit](#)
  - [Cursors](#)
    - [after](#)
    - [orderBy](#)
- [Updating multiple queries](#)
- [Local state](#)
  - [Direct writes](#)
  - [Local mutations](#)
- [REST](#)
- [Review subscriptions](#)
  - [Subscription component](#)
  - [Add new reviews](#)
  - [Update on edit and delete](#)
- [Prefetching](#)
  - [On mouseover](#)
  - [Cache redirects](#)
- [Batching](#)
- [Persisting](#)
- [Multiple endpoints](#)

## Paginating

- [Offset-based](#)
  - [page](#)
  - [skip & limit](#)
- [Cursors](#)
  - [after](#)

- `orderBy`

Our `ReviewsQuery` currently has `limit: 20` because loading all the reviews would be unwise 😊. We don't know how many reviews there will be in the database, and receiving thousands of them over the network would take a long time on mobile. They'd take a lot of memory in the Apollo store, they'd take a long time to render onto the page, and we'd have the problems that come along with a high DOM (and VDOM) node count: interacting with the DOM takes longer, and the amount of memory the browser uses grows—in the worst case, it exceeds the available memory on the device. On mobile, the OS kills the browser process, and on a computer, the OS starts using the hard drive for memory, which is very slow.

😬 So! In any app where the user might want to see a potentially long list of data, we paginate: we request and display a set amount of data, and when the user wants more (either by scrolling down—in the case of infinite scroll—or by clicking a “next page” link or “page 3” link), we request and display more. There are two main methods of pagination: offset-based, which we'll talk about first, and [cursors](#).

We can display the data however we want. The two most common methods are pages (with next/previous links and/or numbered page links like Google search results) and infinite scroll. We can use either data-fetching method with either display method.

## Offset-based

Offset-based pagination is the easier of the two methods to implement—both on the client and on the server. In its simplest form, we request a `page` number, and each page has a set number of items. The Guide server sends 10 items per page, so page 1 has the first 10, page 2 has items 11-20, etc. A more flexible form is using two parameters: `offset` (or `skip`) and `limit`. The client decides how large each page is by setting the `limit` of how many items the server should return. For instance, we can have 20-item pages by first requesting `skip: 0, limit: 20`, then requesting `skip: 20, limit: 20` (“give me 20 items starting with #20”, so items 20-39), then `skip: 40, limit: 20`, etc.

The downside of offset-based pagination is that if the list is modified between requests, we might miss items or see them twice. Take, for example, this scenario:

1. We fetch page 1 with the first 10 items.
2. Some other user deletes the 4th and 5th items.
3. If we were to fetch page 1 again, we would get the new first 10 items, which would now be items 1–3 and 6–12. But we don't refetch page 1—we fetch page 2.
4. Page 2 returns items 13–22. Which means now we're showing the user items 1-10 and 13-22, and we're missing items 11 and 12, which are now part of page 1.

On the other hand, if things are added to the list, we'll see things twice:

1. We fetch page 1 with the first 10 items.
2. Some other user submits two new items.
3. If we were to fetch page 1 again, we would get the 2 new items and then items 1–8. But instead we fetch page 2.
4. Page 2 returns items 9-18, which means our list has items 9 and 10 twice—once from page 1 and once from page 2.

Depending on our application, these issues might never happen, or if they do, it might not be a big deal. If it is a big deal, switching to [cursor-based](#) pagination will fix it. Another possible solution, depending on how often items are added/deleted, is requesting extra pages (to make sure not to miss items) and de-duplicating (to make sure not to display the same item twice). For example, first we could request just page 1, and then when we want page 2, we request both pages 1 and 2. Now if we were in the first scenario above, and the 4th and 5th items were deleted, re-requesting page 1 would get items 11 and 12, which we previously missed. We'll get items 1–3 and 6–10 a second time, but we can match their IDs to objects already in the store and discard them.

Let's see this in action. Normally an API will support a single pagination method, but as we can see from this schema comment, the `reviews` query supports three different methods:

The screenshot shows a GraphQL schema viewer interface. On the left, a sidebar labeled 'SCHEMA' lists various queries and mutations. The 'reviews(...): [Review!]' query is selected and highlighted in blue. To the right of the sidebar, the details for the 'reviews' query are displayed. It shows the query signature: `reviews(limit: Int, page: Int, skip: Int, after: ObjID, orderBy: ReviewOrderBy): [Review!]`. Below this, a note states: 'To paginate, use page, skip & limit, or after & limit'.

## page

If you're jumping in here, `git checkout 16_0.2.0` (tag `16_0.2.0`, or compare `16...17`)

Let's try `page` first. We switch our `ReviewQuery` from using the `limit` parameter to using the `page` parameter, and we use a variable so that `<Reviews>` can say which page it wants.

`src/graphql/Review.js`

```
export const REVIEWS_QUERY = gql`
  query ReviewsQuery($page: Int) {
    reviews(page: $page) {
      ...ReviewEntry
    }
  }
  ${REVIEW_ENTRY}
`
```

`src/components/Reviews.js`

```
const withReviews = graphql(REVIEWS_QUERY, {
  options: { errorPolicy: 'all', variables: { page: 1 } },
```

Now the page displays the first 10 reviews. If we change it to `{ page: 2 }`, we see the second 10 reviews. We could make the page number dynamic, but let's wait to do that with the next method, `skip` and `limit`.

## skip & limit

To use the `skip` and `limit` parameters, we replace `page` with them in the query:

`src/graphql/Review.js`

```
export const REVIEWS_QUERY = gql`
  query ReviewsQuery($skip: Int, $limit: Int) {
    reviews(skip: $skip, limit: $limit) {
      ...ReviewEntry
    }
  }
`
```

```
    ${REVIEW_ENTRY}
```

and update our component:

```
src/components/Reviews.js
```

```
const withReviews = graphql(REVIEWS_QUERY, {
  options: { errorPolicy: 'all', variables: { skip: 0, limit: 10 } },
```

And we see the first 10 reviews. To see the next 10, we can skip the first 10 with `{ skip: 10, limit: 10 }`.

Let's implement infinite scroll, during which the component will provide new values for `skip` when the user scrolls to the bottom of the page. First let's simplify what we're working with by extracting out the list of reviews to

`<ReviewList>`. `<Reviews>` will be left with the header and the add button. Here's our new `<ReviewList>`:

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { graphql } from 'react-apollo'
import { propTypes } from 'graphql-anywhere'

import Review from '../Review'

import { REVIEWS_QUERY, REVIEW_ENTRY } from '../graphql/Review'

class ReviewList extends Component {
  render() {
    const { reviews, loading, user } = this.props

    return (
      <div className="Reviews-content">
        {loading
          ? <div className="Spinner" />
          : reviews.map(review => (
              <Review key={review.id} review={review} user={user} />
            ))
        }
      </div>
    )
  }
}

ReviewList.propTypes = {
  reviews: PropTypes.arrayOf(propType(REVIEW_ENTRY)),
  loading: PropTypes.bool.isRequired,
  user: PropTypes.object
}

const withReviews = graphql(REVIEWS_QUERY, {
  options: { errorPolicy: 'all', variables: { skip: 0, limit: 10 } },
  props: ({ data: { reviews, loading } }) => ({
    reviews,
    loading
  })
})

export default withReviews(ReviewList)
```

We're going to want a spinner at the bottom of the list of reviews to indicate that we're loading more. When the list is really long—as it is in the case of reviews—we don't need to code hiding the spinner, since it's unlikely users will reach the end 😊. Since we'll always have a spinner, we no longer need `loading`:

```
src/components/ReviewList.js
```

```
class ReviewList extends Component {
  render() {
    const { reviews, user } = this.props
```

```

    return (
      <div className="Reviews-list">
        <div className="Reviews-content">
          {reviews && reviews.map(review => (
            <Review key={review.id} review={review} user={user} />
          ))}
        </div>
        <div className="Spinner" />
      </div>
    )
  }
}

ReviewList.propTypes = {
  reviews: PropTypes.arrayOf(PropTypes(REVIEW_ENTRY)),
  user: PropTypes.object
}

const withReviews = graphql(REVIEWS_QUERY, {
  options: { errorPolicy: 'all', variables: { skip: 0, limit: 10 } },
  props: ({ data: { reviews } }) => ({
    reviews
  })
})

```

Note that now we need to guard with `reviews && reviews.map`, since `reviews` is `undefined` during loading.

`graphql()` gives a prop named `data.fetchMore` that we can use to fetch more data using the same query but different variables. Let's use it to create a `loadMoreReviews()` for `ReviewList` to call:

`src/components/ReviewList.js`

```

const withReviews = graphql(REVIEWS_QUERY, {
  options: { errorPolicy: 'all', variables: { skip: 0, limit: 10 } },
  props: ({ data: { reviews, fetchMore } }) => ({
    reviews,
    loadMoreReviews: () => {
      return fetchMore({
        variables: { skip: reviews.length },
        updateQuery: (previousResult, { fetchMoreResult }) => {
          if (!fetchMoreResult.reviews) {
            return previousResult
          }

          return {
            ...previousResult,
            reviews: [
              ...previousResult.reviews,
              ...fetchMoreResult.reviews
            ]
          }
        }
      })
    }
  })
})

```

`variables: { skip: reviews.length }`: we can keep the same `limit` by not including it here. And we know how many to skip for the next query—the amount we currently have, `data.reviews.length`. `updateQuery` is how we tell Apollo to combine the current data with the newly arrived data, which for us is simply putting the new reviews on the end of the `reviews` array. Now we call `loadMoreReviews()` when the user approaches the bottom of the page:

`src/components/ReviewList.js`

```

class ReviewList extends Component {
  componentDidMount() {
    window.addEventListener('scroll', this.handleScroll)
  }
}

```

```

componentWillUnmount() {
  clearTimeout(this.timeoutID)
  window.removeEventListener('scroll', this.handleScroll)
}

handleScroll = (event) => {
  const currentScrollHeight = window.scrollY + window.innerHeight
  const pixelsFromBottom = document.documentElement.scrollHeight - currentScrollHeight
  if (pixelsFromBottom < 250) {
    this.props.loadMoreReviews()
  }
}

```

This works! One issue is that scroll events fire often, so once the user passes the threshold, we're calling `loadMoreReviews()` a lot 🤖. We only need to once, so we want to stop ourselves from calling it again if we just called it. We can tell whether we just called it by looking at `graphql()`'s `data.networkStatus`, which has a numerical value corresponding with different statuses—loading, ready, polling, refetching, etc. It's `3` while Apollo is fetching more data, and then goes back to `7` (ready) when the data has arrived. So we can add in a guard:

```

const FETCH_MORE = 3

class ReviewList extends Component {
  ...

  handleScroll = (event) => {
    if (this.props.networkStatus === FETCH_MORE) {
      return
    }

    const currentScrollHeight = window.scrollY + window.innerHeight
    const pixelsFromBottom = document.documentElement.scrollHeight - currentScrollHeight
    if (pixelsFromBottom < 250) {
      this.props.loadMoreReviews()
    }
  }

  ...

  const withReviews = graphql(REVIEWS_QUERY, {
    options: {
      errorPolicy: 'all',
      variables: { skip: 0, limit: 10 },
      notifyOnNetworkStatusChange: true
    },
  },
  props: ({ data: { reviews, fetchMore, networkStatus } }) => ({
    reviews,
    networkStatus,
    loadMoreReviews: () => {
      ...
    }
  })
)

```

We need to add `networkStatus` to our `props` function to provide it to our component. We also need to set `options.notifyOnNetworkStatusChange` to `true`, which allows `networkStatus` to change to `3`.

Another issue we've got is what happens when someone else adds a review during the time between when the user loads the page and when they scroll to the bottom: `loadMoreReviews()` will query for `reviews(skip: 10, limit: 10)`, which will return items 11-20. However, the 11th item now is the same as the 10th item before, and we already have the 10th item in `previousResult`. When we combine `previousResult` with `fetchMoreResult`, we get a `reviews` array with a duplicated item:

```

updateQuery: (previousResult, { fetchMoreResult }) => {
  if (!fetchMoreResult.reviews) {
    return previousResult
  }

  return {

```

```

    ...previousResult,
    reviews: [
      ...previousResult.reviews,
      ...fetchMoreResult.reviews
    ]
  }
}

```

Since we use the review's `id` for the `key`, React gives us this error in the console:

```
Warning: Encountered two children with the same key
```

We can prevent duplicated objects from being saved in the store by changing `updateQuery`:

```

import find from 'lodash/find'

...

loadMoreReviews: () => {
  return fetchMore({
    variables: { skip: reviews.length },
    updateQuery: (previousResult, { fetchMoreResult }) => {
      if (!fetchMoreResult.reviews) {
        return previousResult
      }

      const newReviews = fetchMoreResult.reviews.filter(
        ({ id }) => !find(previousResult.reviews, { id })
      )

      return {
        ...previousResult,
        reviews: [...previousResult.reviews, ...newReviews]
      }
    }
  })
}

```

We filter out all of the reviews that are already present in `previousResult.reviews`. We can test it out by setting a `skip` that's too low, for instance:

```
variables: { skip: reviews.length - 5 },
```

Now when we scroll down, we should have 15 total reviews on the page instead of 20 and a React duplicate `key` error.

It seems strange at first, but subtracting some number from the length is a good idea to leave in the code! It makes sure—in the case in which some of the first 10 items are deleted—that we don't miss any items. If we still want 10 new items to (usually) show up when we scroll down, then we can also change `limit` to 15:

```
variables: { skip: reviews.length - 5, limit: 15 },
```

The *final* issue is that we get an error when we try to add or delete a review:

```

Error: Can't find field reviews({}) on object (ROOT_QUERY) {
  "chapters": [
    {
      "type": "id",
      "id": "Chapter:-3",
      "generated": false
    },
    ...
  ],
}

```



```

"currentUser": {
  "type": "id",
  "id": "$ROOT_QUERY.currentUser",
  "generated": true
},
"reviews({\"skip\":0,\"limit\":10})": [
  {
    "type": "id",
    "id": "Review:5aa04e9ec3e315449011604c",
    "generated": false
  },
  ...

```

In our `withAddReview` and `withDeleteMutation` HOCs' `update` functions, we're trying to read `REVIEWS_QUERY` from the store. Since we're not specifying variables there, it looks in the store for the root query field `reviews({})`, with no arguments. And we don't have that in our store, because we've never done a `REVIEWS_QUERY` without arguments—we've only done it with a `skip` and `limit`. The error message prints out the current Apollo store's state, and we can see that our `reviews` query has both arguments:

```
"reviews({\"skip\":0,\"limit\":10})": [
```

We need to provide the same arguments to `store.readQuery` so that Apollo knows which field on `ROOT_QUERY` to read from:

`src/components/ReviewForm.js`

```

const withAddReview = graphql(ADD_REVIEW_MUTATION, {
  props: ({ ownProps: { user }, mutate }) => ({
    addReview: (text, stars) => {
      mutate({
        ...
        update: (store, { data: { createReview: newReview } }) => {
          const query = {
            query: REVIEWS_QUERY,
            variables: { skip: 0, limit: 10 }
          }

          const data = store.readQuery(query)
          data.reviews.unshift(newReview)
          store.writeQuery({ ...query, data })
        }
      })
    }
  })
})

```

`src/components/Review.js`

```

const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
    delete: id =>
      mutate({
        ...
        update: store => {
          const query = {
            query: REVIEWS_QUERY,
            variables: { skip: 0, limit: 10 }
          }

          let data = store.readQuery(query)
          remove(data.reviews, { id })
          store.writeQuery({ ...query, data })

          ...
        }
      })
    }
  })
})

```

```
})
```

Now Apollo can successfully read and write to the store, and our optimistic updates will work again.

What happens when `skip` changes to 10 or 20? Do we need to also update our calls to `readQuery`? It turns out that we don't need to—when we call `fetchMore`, the additional results get added to the store under the original root query field. We can see this is the case by scrolling down, opening Apollo devtools -> Cache, and looking at `ROOT_QUERY`:

The screenshot shows the Apollo DevTools interface with the 'Cache' tab selected. On the left, a sidebar lists various cache entries: 'ROOT\_QUERY', 'Chapter:-1', 'Chapter:-2', 'Chapter:-3', 'Chapter:0', 'Chapter:1', 'Chapter:5', 'ROOT\_MUTATION', and several 'Review' entries. The 'ROOT\_QUERY' entry is highlighted. The main panel displays the JSON structure of the 'ROOT\_QUERY' field, which is an object with a 'reviews' property. This property is an array of 20 Review objects, each with a unique ID and a text field. The array is indexed from 0 to 19, and the first few elements are visible in the list.

## Cursors

If you're jumping in here, `git checkout 17_0.2.0` (tag `17_0.2.0`, or compare `17...18`)

Subsections:

- [after](#)
- [orderBy](#)

Cursor-based pagination uses a **cursor**—a pointer to where we are in a list. With cursors, the schema looks different from the Guide schema we've been working with. Our queries could look something like:

```
{
  listReviews (cursor: $cursor, limit: $limit) {
    cursor
    reviews {
      ...ReviewEntry
    }
  }
}
```

Each query comes back with a cursor, which we then include as an argument in our next query. A cursor usually encodes both the ID of the last item and the list's sort order, so that the server knows what to return next. For instance, if the first 10 reviews ended with a review that had an ID of `100`, and the list was ordered by most recently created, the cursor could be `100:createdAt_DESC`, and the query could be:

```
{
  listReviews (cursor: "100:createdAt_DESC", limit: 10) {
    cursor
    reviews {
      ...ReviewEntry
    }
  }
}
```

It would return:

```
{
  "data": {
    "listReviews": {
      "cursor": "90:createdAt_DESC",
      "reviews": [{
        "id": "99"
        ...
      },
      ...
      {
        "id": "90"
        ...
      }
    ]
  }
}
```

And then our next query would be `listReviews (cursor: "90:createdAt_DESC", limit: 10)`.

This is a simple version of cursors. If we're working with a server that follows the [Relay Cursor Connections spec](#) (with `edges` and `nodes` and `pageInfo`s), we can follow [this example](#) for querying it.

## after

Let's implement a version of pagination that has the same information—last ID and sort order—but works within the Guide schema. We can see in [Playground](#) that there are a couple of arguments we haven't used yet—`after` and `orderBy`:

```
enum ReviewOrderBy {
  createdAt_ASC
  createdAt_DESC
}

# To paginate, use page, skip & limit, or after & limit
reviews(limit: Int, page: Int, skip: Int, after: ObjID, orderBy: ReviewOrderBy): [Review!]
```

First, let's use the last review's ID for `after`, and remove `skip`:

`src/components/ReviewList.js`

```
const withReviews = graphql(REVIEWS_QUERY, {
  options: {
    errorPolicy: 'all',
    variables: { limit: 10 },
    notifyOnNetworkStatusChange: true
  },
  props: ({ data: { reviews, fetchMore, networkStatus } }) => ({
```

```

reviews,
networkStatus,
loadMoreReviews: () => {
  if (!reviews) {
    return
  }

  const lastId = reviews[reviews.length - 1].id
  return fetchMore({
    variables: { after: lastId },

```

It's possible that our scroll handler (which calls `loadMoreReviews`) will fire before the results from the initial reviews query has completed, in which case `reviews` will be `undefined`, and we do nothing.

We also have to remove `skip` from `withAddReview` and `withDeleteMutation`, and update the query:

`src/graphql/Review.js`

```

query ReviewsQuery($after: ObjID, $limit: Int) {
  reviews(after: $after, limit: $limit) {

```

It works! And it's so precise that we don't have to worry about things getting added or deleted between `fetchMore` s.

We can even take out that filtering code in `updateQuery` ! [What was the runtime of that thing anyway? It was so big that author Loren was tempted to prematurely optimize with a hash 😊.] One might be concerned about the possibility of the review we're using as a cursor being deleted, but some server implementations cover this case—the Guide API is backed by MongoDB, which has IDs that are comparable based on order of creation, so the server can still find IDs that were created before or after the deleted ID.

## orderBy

Next let's figure out how to get sort order working as well. The two possible values are `createdAt_DESC` (newest reviews first, the default) and `createdAt_ASC`. If we put a “Newest/Oldest” select box in `<Reviews>`, then we can pass the value down to `<ReviewList>` to use in the query's `variables`:

`src/components/Reviews.js`

```

import { MenuItem } from 'material-ui/Menu'
import { FormControl } from 'material-ui/Form'
import Select from 'material-ui/Select'

class Reviews extends Component {
  state = {
    addingReview: false,
    orderBy: 'createdAt_DESC'
  }

  handleOrderByChange = event => {
    this.setState({ orderBy: event.target.value })
  }

  render() {
    ...
    <header className="Reviews-header">
      ...

      <FormControl>
        <Select
          value={this.state.orderBy}
          onChange={this.handleOrderByChange}
          displayEmpty
        >
          <MenuItem value="createdAt_DESC">Newest</MenuItem>
          <MenuItem value="createdAt_ASC">Oldest</MenuItem>
        </Select>
      </FormControl>

```

```

    </header>
  </div>

  <ReviewList user={user} orderBy={this.state.orderBy} />

```

In `<ReviewList>`, we need our `withReviews` HOC to have access to the props. We can use the function form of `options` that we used back in `<Section>`. The function gets the props, from which we take `orderBy`:

`src/components/ReviewList.js`

```

ReviewList.propTypes = {
  reviews: PropTypes.arrayOf(propType(REVIEW_ENTRY)),
  user: PropTypes.object,
  orderBy: PropTypes.string.isRequired
}

const withReviews = graphql(REVIEWS_QUERY, {
  options: ({ orderBy }) => ({
    errorPolicy: 'all',
    variables: { limit: 10, orderBy },
    notifyOnNetworkStatusChange: true
  }),

```

The select input now works—when we change it to “Oldest”, the query variable updates, and a different list of reviews loads. When we go back to “Newest”, the original list immediately appears, because Apollo has that list cached under the original set of variables. We can see in devtools that both lists are indeed cached:

The screenshot shows the Apollo DevTools interface. On the left, the 'Cache' tab is selected, showing a list of cached queries. The main pane displays the JSON data for two queries:

- `reviews({"limit":10,"orderBy":"createdAt_ASC"}): [Review]`
  - 0: Review: 5a6676e7094bf236e215f487
  - 1: Review: 5a6676ec094bf236e215f488
  - 2: Review: 5a8130dfffbf5fc5a4f0323
  - 3: Review: 5a81310bffbfe5fc5a4f0324
  - 4: Review: 5a81311dffbfe5fc5a4f0325
  - 5: Review: 5a813138ffbfe5fc5a4f0326
  - 6: Review: 5a822c6000b4152da30d3578
  - 7: Review: 5a822c7c00b4152da30d3579
  - 8: Review: 5a822c9f00b4152da30d357a
  - 9: Review: 5a822ca900b4152da30d357b
- `reviews({"limit":10,"orderBy":"createdAt_DESC"}): [Review]`
  - 0: Review: 5aa0da982047106b4e181dba
  - 1: Review: 5aa0da672047106b4e181db9
  - 2: Review: 5aa0d9ea2047106b4e181db8
  - 3: Review: 5aa0d9de2047106b4e181db7
  - 4: Review: 5aa0ceec2047106b4e181db6
  - 5: Review: 5aa04e9ec3e315449011604c
  - 6: Review: 5a9baa2ec3e3154490116049
  - 7: Review: 5a9ba88dc3e3154490116048
  - 8: Review: 5a9ba64dc3e3154490116047
  - 9: Review: 5a8b86eb4759e237dc48cc60

But we’ve got the below issue again!

```
Error: Can't find field reviews({"limit":10}) on object (ROOT_QUERY)
```

Whenever we change the variables we’re using with `REVIEWS_QUERY`, we have to change our calls to `readQuery` in our mutation HOCs. First `withAddReview`: what value of `orderBy` do we use? We don’t want a new review to appear at the top of an “Oldest” list—we always want it to appear at the top of the “Newest” list. So we use `createdAt_DESC`:

`src/components/ReviewForm.js`

```
const withAddReview = graphql(ADD_REVIEW_MUTATION, {
  props: ({ ownProps: { user }, mutate }) => ({
    addReview: (text, stars) => {
      mutate({
        ...
        update: (store, { data: { createReview: newReview } }) => {
          const query = {
            query: REVIEWS_QUERY,
            variables: { limit: 10, orderBy: 'createdAt_DESC' }
          }

          const data = store.readQuery(query)

```

Now if we switch to “Oldest”, add a review, and switch back to “Newest”, it will show up at the top. Next is `withDeleteMutation` —we want to remove the review from the current list. We can get the current list order by passing it down from `ReviewList` and then to `delete()` :

`src/components/ReviewList.js`

```
render() {
  const { reviews, user, orderBy } = this.props
  ...
  <Review
    key={review.id}
    review={review}
    user={user}
    orderBy={orderBy}
  />

```

`src/components/Review.js`

```
class Review extends Component {
  ...
  delete = () => {
    this.closeDeleteConfirmation()
    this.props.delete(this.props.review.id, this.props.orderBy).catch(e => {
      if (find(e.graphQLErrors, { message: 'unauthorized' })) {
        alert(' ? You can only delete your own reviews!')
      }
    })
  }
  ...

  const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
    props: ({ mutate }) => ({
      delete: (id, orderBy) =>
        mutate({
          variables: { id },
          optimisticResponse: {
            removeReview: true
          },
          update: store => {
            const query = {
              query: REVIEWS_QUERY,
              variables: { limit: 10, orderBy }
            }

            let data = store.readQuery(query)
            remove(data.reviews, { id })
            store.writeQuery({ ...query, data })

```

## Updating multiple queries

If you're jumping in here, `git checkout 18_0.2.0` (tag `18_0.2.0`). Code from this section isn't included in future tags.

We've been assuming the list of reviews is so long that we can't scroll down enough to reach the end. When we deleted a review, we removed it from the current list, but not the other. Because if it's on Newest and the list is huge, then it won't also be on Oldest. But when we're dealing with smaller lists, it might be on both lists. Let's remove it from both.

We know there will always be a list of reviews in the store ordered by `createdAt_DESC` because that's the default. We don't know whether there will be a list ordered by `createdAt_ASC`. If there isn't, `store.readQuery` will throw an error. So we'll first read and write `DESC`, and then we'll try reading and writing `ASC`:

`src/components/Review.js`

```
const withDeleteMutation = graphql(DELETE_REVIEW_MUTATION, {
  props: ({ mutate }) => ({
    delete: id =>
      mutate({
        variables: { id },
        optimisticResponse: {
          removeReview: true
        },
      },
      update: store => {
        const query = {
          query: REVIEWS_QUERY,
          variables: { limit: 10, orderBy: 'createdAt_DESC' }
        }

        let data = store.readQuery(query)
        remove(data.reviews, { id })
        store.writeQuery({ ...query, data })

        query.variables.orderBy = 'createdAt_ASC'

        try {
          data = store.readQuery(query)
          remove(data.reviews, { id })
          store.writeQuery({ ...query, data })
        } catch (e) {}
      }
    )
  })
})
```

We can also stop passing in `orderBy`.

If we can scroll through our entire list, then any newly added item should go on the end of the Oldest list—not just the beginning of the Newest list as we're doing currently. So let's write the new review to the `orderBy: 'createdAt_ASC'` list:

`src/components/ReviewForm.js`

```
const withAddReview = graphql(ADD_REVIEW_MUTATION, {
  props: ({ ownProps: { user }, mutate }) => ({
    addReview: (text, stars) => {
      mutate({
        ...
        update: (store, { data: { createReview: newReview } }) => {
          const query = {
            query: REVIEWS_QUERY,
            variables: { limit: 10, orderBy: 'createdAt_DESC' }
          }

          let data = store.readQuery(query)
          data.reviews.unshift(newReview)
          store.writeQuery({ ...query, data })

          query.variables.orderBy = 'createdAt_ASC'

          try {
            data = store.readQuery(query)
            data.reviews.push(newReview)
            store.writeQuery({ ...query, data })
          } catch (e) {}
        }
      })
    }
  })
})
```

That wasn't too hard—we're updating two queries when deleting and two when creating. But imagine if we had more sort orders—like last updated or number of stars—and added filters—like number of stars or sentences:

```
# orderBy
createdAt_DESC
createdAt_ASC
updatedAt_DESC
updatedAt_ASC
stars_DESC
stars_ASC

# minStars
1
2
3
4
5

# minSentences
1
2
3
4
5
```

Now when there's a change, there are many more lists in the cache we might need to update:

```
orderBy: createdAt_DESC, minStars: 1, minSentences: 1
orderBy: createdAt_DESC, minStars: 1, minSentences: 2
orderBy: createdAt_DESC, minStars: 1, minSentences: 3
orderBy: createdAt_DESC, minStars: 1, minSentences: 4
orderBy: createdAt_DESC, minStars: 1, minSentences: 5
orderBy: createdAt_DESC, minStars: 2, minSentences: 1
orderBy: ...
```

In total, there are  $6 \times 5 \times 5 = 150$  possibilities. That's a lot of updating to do. Not only that, but we have to add in some logic—for instance, if a new review is submitted with 4 sentences and 4 stars, we don't want to also add it to any of the `minStars: 5` or `minSentences: 5` filtered lists.

When our cache-updating code gets this complicated, we can use [apollo-link-watched-mutation](#) to simplify things. With it, we define a function that, given the mutation and the query, returns a new query result. And the package takes care of calling our function as many times as it needs to (for as many variations of the query variables exist in the store) as well as reading from and writing to the store. We won't actually use this in our app, but here's how we would implement it:

```
npm install apollo-link-watched-mutation
```

src/index.js :

```
import { ApolloLink } from 'apollo-link'

import { getWatchedMutationLink } from '../lib/watchedMutationLink'

const cache = new InMemoryCache()

const link = ApolloLink.from([
  getWatchedMutationLink(cache),
  errorLink,
  networkLink
])
```



lib/watchedMutationLink.js :

```
import WatchedMutationLink from 'apollo-link-watched-mutation'

export const getWatchedMutationLink = cache =>
  new WatchedMutationLink(cache, {
    DeleteReview: {
      ReviewsQuery: ({ mutation, query: { result } }) => {
        ...
      },
      UserQuery: ({ mutation, query: { result } }) => {
        ...
      }
    },
    AddReview: {
      ReviewsQuery: ({ mutation, query }) => {
        ...
      }
    }
  })
```

We would also remove the `update` functions from our `mutate()` calls in our `withAddReview` and `withDeleteMutation` HOCs.

The first arg to the `WatchedMutationLink` constructor is our cache, and the second is an object that lists each mutation and the queries we want to update when that mutation is complete. In this case, the mutation named `DeleteReview` updates `ReviewsQuery` and `UserQuery`, and the `AddReview` mutation updates `ReviewsQuery`. At first, looking at our deletion update function, we might think to update `ReviewsQuery` and `ReadUserFavorites`:

src/components/Review.js

```
const READ_USER_FAVORITES = gql`
  query ReadUserFavorites {
    currentUser {
      id
      favoriteReviews {
        id
      }
    }
  }
`

...

update: store => {
  const query = {
    query: REVIEWS_QUERY,
    variables: { limit: 10, orderBy: 'createdAt_DESC' }
  }

  let data = store.readQuery(query)
  remove(data.reviews, { id })
  store.writeQuery({ ...query, data })

  query.variables.orderBy = 'createdAt_ASC'

  try {
    data = store.readQuery(query)
    remove(data.reviews, { id })
    store.writeQuery({ ...query, data })
  } catch (e) {}

  data = store.readQuery({ query: READ_USER_FAVORITES })
  remove(data.currentUser.favoriteReviews, { id })
  store.writeQuery({ query: READ_USER_FAVORITES, data })
}
```

However, `ReadUserFavorites` is only used to write data to the store in this update function—it is not a *watched query* (a watched query is a query attached to a React component with a `<Query>` or a `graphql()` HOC).

`WatchedMutationLink` goes through all the watched queries and calls functions of the same name in our config, so if we had such a function, it would never get called:

```
new WatchedMutationLink(cache, {
  DeleteReview: {
    ReviewsQuery: ({ mutation, query: { result } }) => {
      // this function gets called
    },
    ReadUserFavorites: ({ mutation, query: { result } }) => {
      // this function never gets called
    }
  }
})
```

What we're trying to update is the `favoriteCount` in the `<Reviews>` header, and that's calculated from the `user` prop, which is fetched by the `UserQuery`, so we'll add a `UserQuery` update function to our config object. Here are the `ReviewsQuery` and `UserQuery` functions filled in:

```
import remove from 'lodash/remove'

new WatchedMutationLink(cache, {
  DeleteReview: {
    ReviewsQuery: ({ mutation, query: { result } }) => {
      const idBeingDeleted = mutation.variables.id
      remove(result.reviews, {
        id: idBeingDeleted
      })
      return result
    },
    UserQuery: ({ mutation, query: { result } }) => {
      const idBeingDeleted = mutation.variables.id
      remove(result.currentUser.favoriteReviews, { id: idBeingDeleted })
      return result
    }
  }
})
```

The update function is given the `mutation` that just occurred and the `query` that is being watched. The `query.result` is the current data in the store—in the case of `ReviewsQuery`, a list of `reviews`. The value we return from the function is the updated result, and contains what we want written back to the store. Here we remove the deleted review from `result.reviews` and return `result`.

Each function gets called as many times as there are matching queries. For example, if there were all 150 combinations of possible `ReviewsQuery` arguments in the store, our `ReviewsQuery()` function would be called 150 times, and the value of `result.reviews` would be a different list of reviews each time.

Updating after `AddReviews` is more complicated. We don't have to worry about the `favoriteCount`, because a new review hasn't been favorited, but the logic for updating `ReviewsQuery` lists is much longer than our above removal logic:

```
import findIndex from 'lodash/findIndex'

new WatchedMutationLink(cache, {
  DeleteReview: { ... },
  AddReview: {
    ReviewsQuery: ({ mutation, query }) => {
      const newReview = mutation.result.data.createReview

      const { orderBy, minStars, minSentences } = query.variables
      const { reviews } = query.result

      const countSentences = ({ text }) => text.match(/\w[.?!](\s|$)/g).length
      if (
        newReview.stars < minStars ||
```

```

    countSentences(newReview) < minSentences
  ) {
    // don't add to store
    return
  }

  switch (orderBy) {
    case 'createdAt_DESC':
    case 'updatedAt_DESC':
      reviews.unshift(newReview)
      break
    case 'createdAt_ASC':
    case 'updatedAt_ASC':
      reviews.push(newReview)
      break
    case 'stars_DESC':
      const insertBefore = findIndex(
        reviews,
        review => review.stars <= newReview.stars
      )
      reviews.splice(insertBefore - 1, 0, newReview)
      break
    case 'stars_ASC':
      const insertBefore = findIndex(
        reviews,
        review => review.stars >= newReview.stars
      )
      reviews.splice(insertBefore - 1, 0, newReview)
      break
  }

  // return the query result, including the modified query.result.reviews
  return query.result
}
})

```

Depending on what the query variables are ( `orderBy` , `minStars` , and `minSentences` ), we have to decide whether to add the new review to the list, and if yes, where to add it. If the new review's number of stars or sentences is below the minimum, we return `undefined` so that `WatchedMutationLink` knows to not alter the store. If the sort order is by `createdAt` or `updatedAt` , we add it to the top or bottom of the list. And if the sort order is by number of stars, we go through the list to find the right location.

## Local state

Section contents:

- [Direct writes](#)
- [Local mutations](#)

In most of the apps we build, the majority of our *state* (the data that backs our UI) is *remote state*—it comes from a server and is saved in a database. But some of our state doesn't come from the server and isn't stored in a database—it originates locally on the client and stays there. This type of data is called our *local state*. One example of local state is a user setting that for whatever reason we didn't want to send to the server to persist. Another example is data from a device API: if we were making a navigation app, we would want to display the device's location and speed. A simple solution would be to put the state in a variable, for instance `window.gps` :

```

navigator.geolocation.watchPosition(position => {
  window.gps = position.coords
})

```

And then we'd reference that variable when we needed it. However, there are a couple of issues with this solution. One is that we'd like be able to trigger view updates when the data changes. We could move it to a component's `this.state`, but A) when the component is unmounted, we lose the data, and B) if we need the data in different places in the app, we'd have to pass it around as a prop a lot. The other issue is the lack of structure—with a large app and many developers, it gets hard to know what state is out there in variables scattered around the codebase, the data format of each variable, and how each should be modified. A popular solution that addresses both of these issues is [Redux](#), a library for maintaining global state.

*Global state* means state accessible from anywhere in your app, as opposed to *component state*, which is accessible as `this.state` inside the component in which it's created, or as a prop if the data is passed to children. **Global vs component** state is tangential to **local vs remote** state. The former is about where on the client the state is kept, and the latter is about whether or not the data is stored on the server.

Redux provides a structure for reading and modifying data, and it re-renders components when the data changes. While Redux is great, Apollo has its own solution to local state which addresses the same issues. Choosing the system we're already using for local state will make it simpler to implement and result in more understandable, concise code.

Apollo stores local state alongside remote state in the Apollo store. One guess as to how we read it out of the store... with GraphQL queries! All we do is add a `@client` [directive](#):

```
query LocationQuery {
  gps @client {
    lat
    lng
  }
}
```

There are two ways to update our local state—direct writes or mutations. Direct writes are easy to do—we just call the `apollo.writeData()` function, which writes data to the store. Mutations take some work to set up, but provide structure that's useful in a large codebase or when working with a team. They also allow for reading from the store, which we need to do sometimes in order to figure out what to write.

## Direct writes

If you're jumping in here, `git checkout 18_0.2.0` (tag [18\\_0.2.0](#), or compare [18...19](#))

`apollo.writeData()` simply takes an object with a `data` property and writes the data to the store at the root level. So if we wanted to be able to make the root `gps @client` query above, we would do:

```
navigator.geolocation.watchPosition(position => {
  apollo.writeData({ data: { gps: position.coords } })
})
```

A place in our app where a simple piece of local state would be useful is during login. Right now, our `withUser()` HOC provides a `loggingIn` boolean that's true when the `currentUser` query is `loading`. But it would be more accurate if `loggingIn` were true as soon as the user clicks the "Sign in" button. If we had a piece of state called `loginInProgress` that was true while the user went through the Auth0 login process, then we could update `loggingIn` to be `loading || loginInProgress`:

`src/lib/withUser.js`

```
export const USER_QUERY = gql`
  query UserQuery {
    currentUser {
      id
      firstName
      name
    }
  }
}`
```

```

      username
      email
      photo
      hasPurchased
      favoriteReviews {
        id
      }
    }
  }
  loginInProgress @client
}
}

export const withUser = graphql(USER_QUERY, {
  props: ({ data: { currentUser, loading, loginInProgress } }) => ({
    user: currentUser,
    loggingIn: loading || loginInProgress
  })
})
})

```

We update with `writeData()` :

`src/lib/auth.js`

```

export const login = () => {
  apollo.writeData({ data: { loginInProgress: true } })

  auth0Login({
    onCompleted: e => {
      apollo.writeData({ data: { loginInProgress: false } })

      if (e) {
        console.error(e)
        return
      }

      apollo.refetchObservableQueries()
    }
  })
}

```

We make sure to set it to `false` first thing in `onCompleted` so that it runs even when there's an error. Lastly, we need to set up local state:

`src/lib/apollo.js`

```

export const apollo = new ApolloClient({ link, cache, resolvers: {} })

const initializeCache = () => {
  cache.writeData({
    data: {
      loginInProgress: false
    }
  })
}

initializeCache()

apollo.onResetStore(initializeCache)

```

First we add a `resolvers` object to `ApolloClient` so that it knows we're using local state. Then we make a function to initialize the cache by setting `loginInProgress` to `false`, and we call the function both on pageload and when the store is reset.

Now it's working—when we click the “Sign in” button, we can see the spinner while the Auth0 popup is open 😊.

## Local mutations

If you're jumping in here, `git checkout 19_0.2.0` (tag `19_0.2.0`, or compare `19...20`)

Now let's add a local-state feature that uses a mutation. Currently, whenever we switch between sections, one of two things happens to our scroll position:

- If we don't have the section content on the client, `scrollTop` is set to 0 when the loading skeleton is displayed.
- If we do have the section content on the client, `scrollTop` remains the same.

It would be nice for the user if, when switching back to a section they were previously reading, the scroll position updates to where they were. So let's save their last position for each section in local state! We could implement this feature with direct writes, but let's see what the more structured method—a local mutation—looks like.

When implementing a new feature, the best place to start is often the schema. Thus far we've been using an existing schema defined on the `api.graphql.guide` server. But we're writing a client-side mutation, so the schema for it will live on the client, and Apollo will combine the client and server schemas into one, which we'll be able to see in the devtools.

Let's call the mutation `setSectionScroll`, and all it needs is the section id and `scrollTop`:

```
type Mutation {
  setSectionScroll(id: String!, scrollTop: Int!): Boolean
}
```

Normally a mutation that updates a type will return that type—in this case a `Section`:

```
type Mutation {
  setSectionScroll(id: String!, scrollTop: Int!): Section
}
```

And then we'd request the `id` and new `scrollTop` so that Apollo could update the store:

```
mutation {
  setSectionScroll(id: "foo", scrollTop: 10) {
    id
    scrollTop
  }
}
```

But this mutation will be run on the client, and it will be modifying the store. So all we need to return from the mutation is `true` to indicate success.

We pass our client-side schema to `ApolloClient()`:

`src/lib/apollo.js`

```
import gql from 'graphql-tag'

const typeDefs = gql`
  type Query {
    loginInProgress: Boolean
  }
  type Mutation {
    setSectionScroll(id: String!, scrollTop: Int!): Boolean
  }
`

export const apollo = new ApolloClient({
  link,
  cache,
  typeDefs,
  resolvers: {}
})
```

Every schema needs to define a `Query` type, so we put our top-level `loginInProgress` field from the [last section](#). Next let's write the `setSectionScroll` resolver:

```
export const apollo = new ApolloClient({
  link,
  cache,
  typeDefs,
  resolvers: {
    Mutation: {
      setSectionScroll: (_, { id, scrollY }, { cache, getCacheKey }) => {
        const cacheKey = getCacheKey({ __typename: 'Section', id })
        cache.writeData({ id: cacheKey, data: { scrollY } })
        return true
      }
    }
  }
})
```

We're given the `getCacheKey()` function which generates the *store ID* we talked about in the [Arbitrary updates](#) section—for example `'Section:1-1'`. Then we write the new `scrollY` to the cache (just another term for the store).

Now let's add a `<Mutation>` component so that we can call the mutation:

`src/components/Section.js`

```
Section.propTypes = {
  ...
  setScrollPosition: PropTypes.func.isRequired
}

const SET_SECTION_SCROLL_MUTATION = gql`
  mutation SetSectionScroll($id: String!, $scrollY: Int!) {
    setSectionScroll(id: $id, scrollY: $scrollY) @client
  }
`

const SectionWithData = ({ location: { state, pathname } }) => {
  ...
  <Mutation mutation={SET_SECTION_SCROLL_MUTATION}>
    {setScrollPosition => (
      <Section
        {...createProps(queryInfo)}
        viewedSection={viewedSection}
        setScrollPosition={setScrollPosition}
      />
    )}
  </Mutation>
}
```

Note the `@client` in the document: `setSectionScroll(id: $id, scrollY: $scrollY) @client`. It tells the state link to resolve the mutation on the client instead of passing it on to the server.

Now we need `<Section>` to call `setScrollPosition()`, which means we need a scroll handler:

```
import debounce from 'lodash/debounce'

class Section extends Component {
  ...

  componentDidMount() {
    window.addEventListener('scroll', this.handleScroll)

    if (this.props.section) {
      this.viewedSection(this.props.section.id)
    }
  }

  componentWillUnmount() {
```

```

clearTimeout(this.timeoutID)
window.removeEventListener('scroll', this.handleScroll)
}

handleScroll = debounce(() => {
  if (this.props.section.scrollY === window.scrollY) {
    return
  }

  this.props.setScrollPosition({
    variables: {
      id: this.props.section.id,
      scrollY: window.scrollY
    }
  })
}, 1000)
...
}

```

When the component mounts, we add the scroll listener, and on unmount, we remove the listener. For performance, we `debounce` the listener (we prevent it from being called continuously, waiting until the user has stopped scrolling for a second). Then inside the listener, we call the mutation (first checking to make sure the `scrollY` has changed).

It now works! Well, at least the mutation gets called. Which we wouldn't know was happening correctly unless we looked at the devtools mutation log and cache:

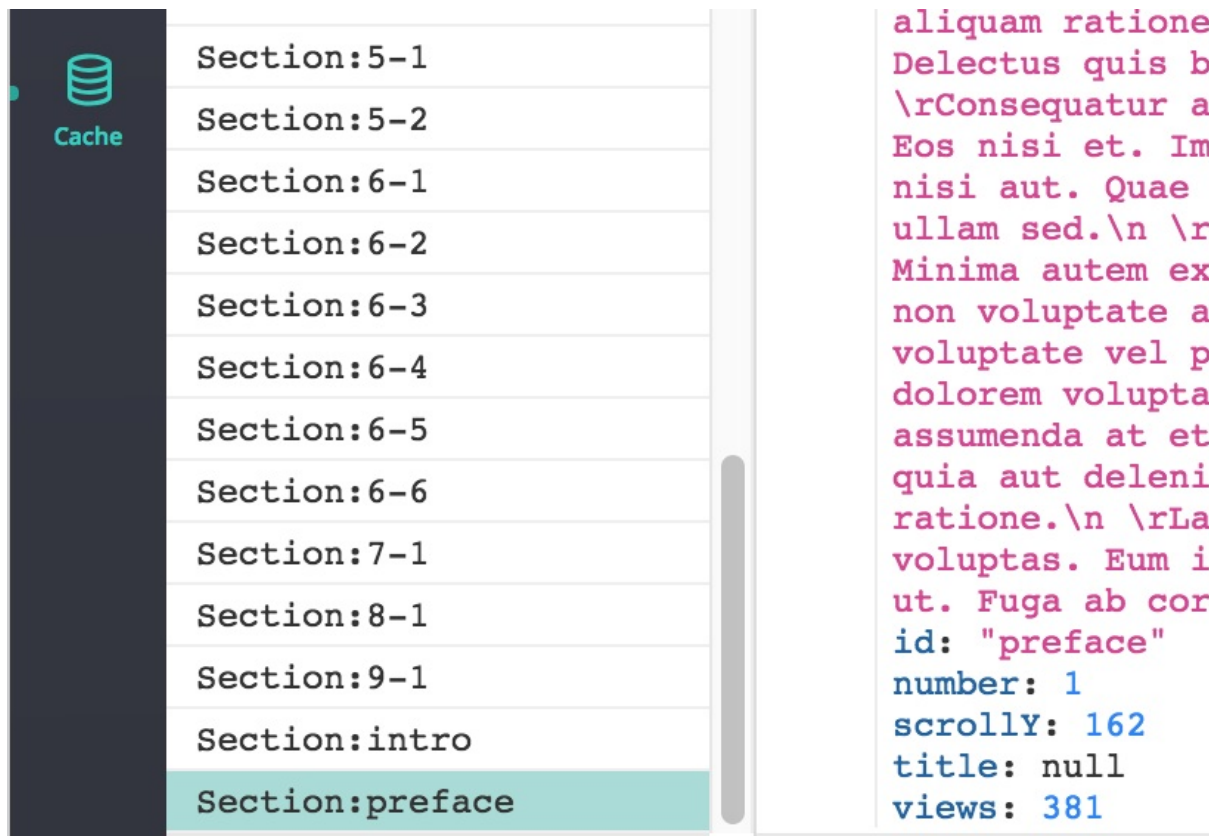
The screenshot shows the GraphQL DevTools interface. On the left, a sidebar contains icons for GraphQL, Queries, and Mutations. The 'Mutation log' panel is active, displaying a list of four mutations: 1. ViewedSection, 2. SetSectionScroll, 3. SetSectionScroll, and 4. SetSectionScroll. The details for the selected 'SetSectionScroll' mutation are shown on the right. It includes a 'Run in GraphQL' button, a 'Variables' section with 'id' set to 'preface' and 'scrollY' set to 162, and a 'Mutation string' section containing the following GraphQL code:

```

mutation SetSectionScroll($id: String!, $scrollY: Int!) {
  setSectionScroll(id: $id, scrollY: $scrollY) @client
}

```





What's left is: when the section changes, updating the scroll position to the new section's `scrollY`. For that, we have to add it to our section queries:

```
Section.propTypes = {
  section: PropTypes.shape({
    ...
    scrollY: PropTypes.number
  }),
  ...
}

const SECTION_BY_ID_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      id
      content
      views
      scrollY @client
    }
  }
`

const SECTION_BY_CHAPTER_TITLE_QUERY = gql`
  query SectionByChapterTitle($title: String!) {
    chapterByTitle(title: $title) {
      title
      section(number: 1) {
        id
        content
        views
        scrollY @client
      }
    }
  }
`

const SECTION_BY_NUMBER_QUERY = gql`
  query SectionByNumber($chapterNumber: Int!, $sectionNumber: Int!) {
    chapterByNumber(number: $chapterNumber) {
```

```

      number
      section(number: $sectionNumber) {
        id
        number
        title
        content
        views
        scrollY @client
      }
    }
  }
}

const SectionWithData = ({ location: { state, pathname } }) => {
  ...

  if (state) {
    query = SECTION_BY_ID_QUERY
    variables = { id: state.section.id }
    createProps = ({ data, loading }) => ({
      section: {
        ...state.section,
        content: get(data, 'section.content'),
        views: get(data, 'section.views'),
        scrollY: get(data, 'section.scrollY')
      },
      chapter: state.chapter,
      loading
    })
  }
  ...
}

```

And then use the new `section.scrollY` prop inside of `<Section>` :

```

class Section extends Component {
  ...

  updateScrollPosition = () => {
    window.scrollTo(0, this.props.section.scrollY)
  }

  componentDidMount() {
    window.addEventListener('scroll', this.handleScroll)

    if (this.props.section) {
      this.viewedSection(this.props.section.id)
      this.updateScrollPosition()
    }
  }

  componentDidUpdate(prevProps) {
    if (!this.props.section) {
      return
    }

    const { id } = this.props.section
    const sectionChanged = get(prevProps, 'section.id') !== id

    if (sectionChanged) {
      this.viewedSection(id)
      this.updateScrollPosition()
    }
  }
  ...
}

```

We call our new `updateScrollPosition()` method when we first mount and when the section changes. If we look back at the browser, we'll see a lot of errors. Above the errors is a warning:

```
Missing field scrollY in {
  "id": "1-7",
  "content": ...
}
```

Apollo is trying to read `scrollY` from a section in the cache and can't find it. We can fix this by giving a default value for `scrollY`, which we do in a resolver:

`src/lib/apollo.js`

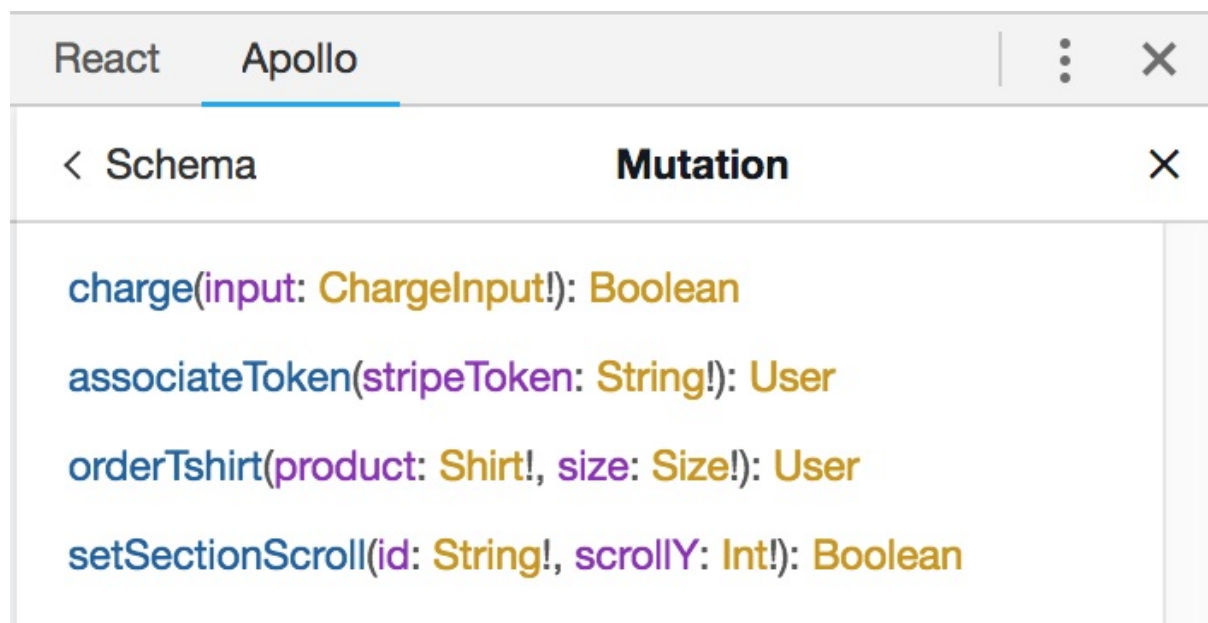
```
export const apollo = new ApolloClient({
  ...
  resolvers: {
    Section: {
      scrollY: () => 0
    },
    Mutation: { ... }
  }
})
```

We can't use `cache.writeData()` like we did with `loginInProgress` because `scrollY` isn't a single field at the top level—we need it to default to 0 for any `Section` object.

Now Apollo will first look in the store for a `Section`'s `scrollY`, and if it's not there, fall back to the resolver that always returns zero.

We can test to see if everything's working: when we load the preface, scroll down, and click "Introduction", we see that the scroll position is at the top, and then when we click "Preface", the scroll position goes back down to where we were!

Looking back, we can see how local mutations provided more structure than direct writes, which could be helpful down the road. Our teammates or our future (forgetful) selves can look at our `SET_SECTION_SCROLL_MUTATION` and see what function and arguments that `<Mutation>` provides, whereas with direct writes, we might have to look for how a piece of local state is used (read from the store) in order to figure out what to write to the store. Our future selves can also look at the `withClientState()` call in order to see all the options for local mutations, or can look at devtools to see the entire schema, with local mutations included at the bottom of the `Mutation` type:



The other reason to use local mutations is when we need to read from the store in order to write. We didn't need to with `setSectionScroll`, but if we had a `clapForSection` mutation, we would:

```
export const apollo = new ApolloClient({
  ...
  resolvers: {
    ...
    Mutation: {
      clapForSection: (_, { id }, { cache, getCacheKey }) => {
        const cacheKey = getCacheKey({ __typename: 'Section', id })
        const fragment = gql`
          fragment clap on Section {
            claps
          }
        `
        const section = cache.readFragment({ fragment, id: cacheKey })

        cache.writeFragment({
          fragment,
          id: cacheKey,
          data: {
            claps: section.claps + 1
          }
        })
        return true
      }
    }
  }
})
```

Here we read a fragment containing `claps`, the field we need, and then we write to the store the previous `claps` plus 1. We're using `cache.writeFragment`, which is like `cache.writeData`, but validates the `data` argument to make sure it matches the `fragment` argument.

## REST

If you're jumping in here, `git checkout 20_0.2.0` (tag `20_0.2.0`, or compare `20...21`)

You might be thinking, “What is a section on REST doing in a chapter on GraphQL client dev??” The thing is, not all of our colleagues have seen the light of GraphQL yet, so they're still making REST APIs! 😞 And we might want to use them in our app. The common solution is for your backend GraphQL server to proxy the REST API. For example, the server will add a query to the schema:

```
type Query {
  githubStars
  ...
  latestSatelliteImage(lon: Float!, lat: Float!, sizeInDegrees: Float): String
}
```

And we would write our client query:

```
query WhereAmI {
  latestSatelliteImage(lon: -73.94, lat: 40.7, sizeInDegrees: 0.3)
}
```

And when the server received our query, it would send this GET request to NASA:

[https://api.nasa.gov/planetary/earth/imagery/?lon=-73.94&lat=40.7&dim=0.3&api\\_key=DEMO\\_KEY](https://api.nasa.gov/planetary/earth/imagery/?lon=-73.94&lat=40.7&dim=0.3&api_key=DEMO_KEY)

The server would get back a URL of an image, which it would return to us, which we would put in the `src` of an `<img>` tag:



So that's how proxying through our GraphQL backend works (and we'll go into more detail in the server chapter). But what if our backend can't proxy the REST API? Maybe we don't have control over the backend, or maybe some less common reason applies, like needing to reduce load on the server or needing better latency (proxying through the server is slightly slower). In that case, we can use [apollo-link-rest](#) to send some of our GraphQL queries as REST requests to a REST API instead of to our GraphQL server!

We need to find a REST API to use in our Guide app so that we can learn by example in this section of the book 🤔. Displaying a satellite image isn't useful, but displaying the temperature in the header might conceivably be useful (albeit completely unrelated to GraphQL 😊). If we google "weather api", the first result is OpenWeatherMap, and we see that it's free to use—great. Now we want to open up Playground to look at the OpenWeatherMap's schema to figure out which query to use. But it's a REST API! And REST doesn't have a specified way of reporting what the API can do, so we can't have a standard tool like Playground that shows us. So we have to read their docs. Let's use their [current weather data](#) endpoint, `api.openweathermap.org/data/2.5/weather`, which looks like it has a number of options for specifying the location with query parameters:

- `?q=[city name]`
- `?id=[city id]`
- `?lat=[latitude]&lon=[longitude]`
- `?zip=[zip code]`

Which one can we use? We don't know the client's city or GPS coordinates or zip code... so at the moment, none of them! There are a couple of ways, though, to get the user's location:

- Query an IP geolocation API, which looks up the client's IP in a database and returns that IP's city and approximate coordinates.

- Use the web standard [geolocation API](#), which according to [caniuse](#) works in all browsers after IE 8! Except for Opera Mini 😊.

The browser API is more precise, easier to code, and gets the user's consent via a built-in permission dialog. So let's do that. All we need to do is just `navigator.geolocation.getCurrentPosition`, and after the user approves, we get the coordinates in a callback:

```

window.navigator.geolocation.getCurrentPosition(
  ({ coords: { latitude, longitude } }) => {
    console.log(latitude, longitude)
    // logs: 40.7 -73.94
  }
)

```

Now we have numbers to put into our URI format, which was:

```
api.openweathermap.org/data/2.5/weather?lat=[latitude]&lon=[longitude]
```

And we also need an API key, which their docs say should go in an `appid` query param. The full URL, broken down:

```

http://
api.openweathermap.org
/data/2.5/weather
?lat=40.7
&lon=-73.94
&appid=4fb00091f111862bed77432aead33d04

```

And the link:

<http://api.openweathermap.org/data/2.5/weather?lat=40.7&lon=-73.94&appid=4fb00091f111862bed77432aead33d04>

If this API key is over its limit, you can [get a free one here](#).

We get a response like this:

```

{
  "coord": { "lon": -73.94, "lat": 40.7 },
  "weather": [
    {
      "id": 803,
      "main": "Clouds",
      "description": "broken clouds",
      "icon": "04n"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 283.59,
    "pressure": 1024,
    "humidity": 66,
    "temp_min": 280.95,
    "temp_max": 285.95
  },
  "visibility": 16093,
  "wind": { "speed": 2.26, "deg": 235.503 },
  "clouds": { "all": 75 },
  "dt": 1539575760,
  "sys": {
    "type": 1,
    "id": 2121,
    "message": 0.0044,
    "country": "US",
    "sunrise": 1539601626,
    "sunset": 1539641711
  },
  "id": 5125125,
  "name": "Long Island City",

```

```
"cod": 200
}
```

That's a lot of stuff. Since it's not GraphQL, we didn't know what we were going to get back until we tried it, unless we were able to find it in their docs (which, eventually, author Loren did—under the heading “Weather parameters in API respond”). Looking through the response JSON, we find `main.temp`, which is a weirdly high number, so we might suspect it's Kelvin, and we can search the docs to confirm. (In a GraphQL API, this could have been included in a schema comment, and we wouldn't have had to search 😊.)

If we didn't have Apollo, we would use `fetch()` or `axios.get()` to make the HTTP request:

```
const weatherEndpoint = 'http://api.openweathermap.org/...'
const response = await fetch(weatherEndpoint)
const data = await response.json();
console.log(`It is ${data.main.temp} degrees Kelvin`);
```

### Run in browser

And we would use component lifecycle methods and `setState` to get the returned data into our JSX. Or if we wanted the data cached so that we can use it in other components or on future instances of the current component, or if we wanted all of our data fetching logic separated from our presentational components, we might use `Redux` instead.

However, with `apollo-link-rest` we can get Apollo to make the HTTP request for us, cache the response data for future use, and provide the data to our components. First, we set up the link:

`src/lib/apollo.js`

```
import { ApolloLink, split } from 'apollo-link'
import { RestLink } from 'apollo-link-rest'

...

const restLink = new RestLink({
  uri: 'https://api.openweathermap.org/data/2.5/'
})

const link = ApolloLink.from([errorLink, restLink, networkLink])
```

Since requests flow from left to right in the link chain, we want our `restLink` to be to the left of `networkLink` (it won't pass on REST requests to `networkLink`, which would send them to our GraphQL server). And since responses (and errors) flow from right to left, we want `restLink` to be to the right of `errorLink` so that errors from `restLink` go through `errorLink`.

Let's add a temperature component in the header:

`src/components/App.js`

```
import CurrentTemperature from './CurrentTemperature'

...

<header className="App-header">
  <StarCount />
  <Link ... />
  <CurrentUser />
  <CurrentTemperature />
</header>
```

And now for its implementation. Let's start with the query:

```
{
  weather(lat: $lat, lon: $lon)
```



```

    @rest(
      type: "WeatherReport"
      path: "weather?appid=4fb00091f111862bed77432aead33d04&{args}"
    ) {
      main
    }
  }
}

```

Anything with the `@rest` directive `apollo-link-rest` will resolve itself. We've already configured the link with the base of the URI, so here we give the rest of it. Since we're getting back an object, we also need to make up a name for what the object's type will be in the Apollo store. And we want the `"main"` attribute from the JSON response, so `{ main }` is our selection set.

If we want to be even more explicit about which data we're using, we could select just `main.temp` instead of the whole `main` object. But when we want to select fields in objects, we need the object to have a type, so we add an `@type` directive:

```

query TemperatureQuery {
  weather(lat: $lat, lon: $lon)
    @rest(
      type: "WeatherReport"
      path: "weather?appid=4fb00091f111862bed77432aead33d04&{args}"
    ) {
    main @type(name: "WeatherMain") {
      temp
    }
  }
}

```

Now let's think about the UX. At some point we need to call `window.navigator.geolocation.getCurrentPosition`, after which the browser prompts the user to share their location. We don't want to annoy users with this prompt every time they use the app, so let's start out with a button and go through these steps:

- Display location button
- User clicks button and we request their location from the browser
- User gives permission through browser dialog
- We receive the location and make the query
- We receive the query results and display them

Here's the shell of our component with that logic and our lat/lon state:

`src/components/CurrentTemperature.js`

```

import React, { Component } from 'react'
import { Query } from 'react-apollo'
import gql from 'graphql-tag'
import IconButton from '@material-ui/core/IconButton'
import MyLocation from '@material-ui/icons/MyLocation'

class CurrentTemperature extends Component {
  state = {
    lat: null,
    lon: null
  }

  requestLocation = () => { ... }

  render() {
    const dontHaveLocationYet = !this.state.lat

    return (
      <div className="Weather">
        <Query
          query={TEMPERATURE_QUERY}
          skip={dontHaveLocationYet}

```



```

    variables={{ lat: this.state.lat, lon: this.state.lon }}
  >
  <Query>
    {({ data, loading }) => {
      if (dontHaveLocationYet) {
        return (
          <IconButton
            className="Weather-get-location"
            onClick={this.requestLocation}
            color="inherit"
          >
            <MyLocation />
          </IconButton>
        )
      }

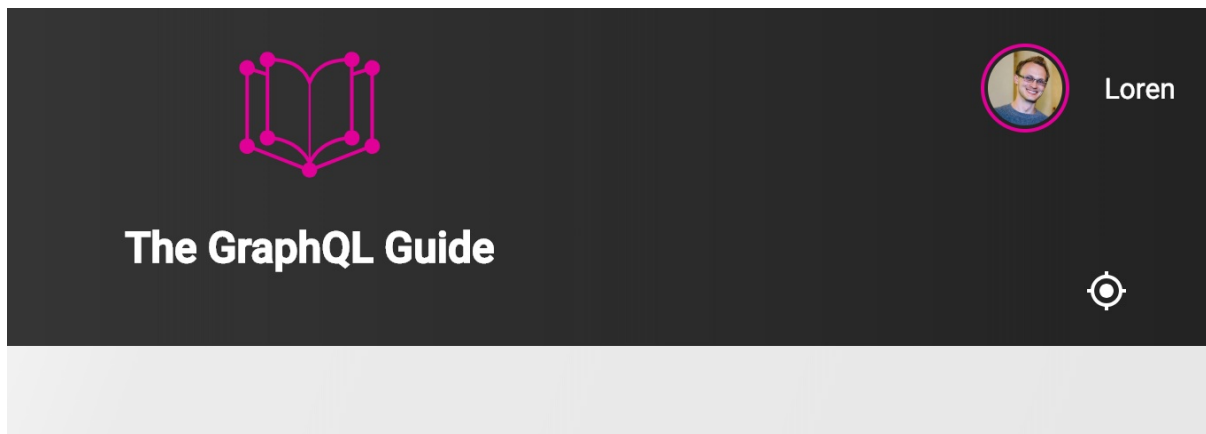
      return data.weather.main.temp
    }}
  </Query>
</div>
)
}
}

const TEMPERATURE_QUERY = gql`
  query TemperatureQuery {
    weather(lat: $lat, lon: $lon)
    @rest(
      type: "WeatherReport"
      path: "weather?appid=4fb00091f111862bed77432ae4d33d04&{args}"
    ) {
      main
    }
  }
`

export default CurrentTemperature

```

When we don't yet have the user's location, we skip running the query and show the location button. Once we do have the location, we pass it to our query and display `data.weather.main.temp`.



It would be nice to display a spinner while we're waiting for the location and the weather API, so let's fill in `requestLocation()` and add `gettingPosition` to the state:

```

class CurrentTemperature extends Component {
  state = {
    lat: null,
    lon: null,
    gettingPosition: false
  }

  requestLocation = () => {
    this.setState({ gettingPosition: true })
    window.navigator.geolocation.getCurrentPosition(

```

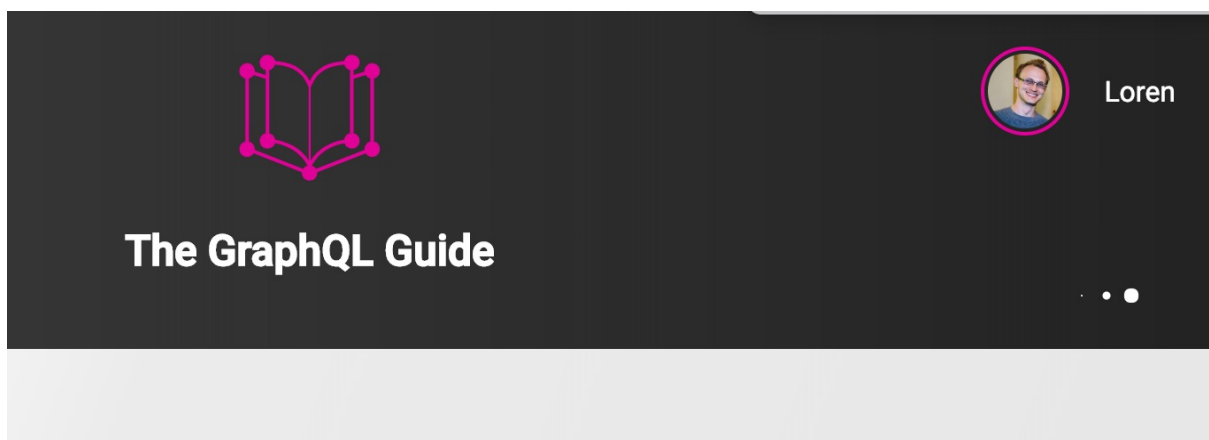
```

    ({ coords: { latitude, longitude } }) => {
      this.setState({ lat: latitude, lon: longitude, gettingPosition: false })
    }
  }
}

render() {
  const dontHaveLocationYet = !this.state.lat

  return (
    <div className="Weather">
      <Query
        query={TEMPERATURE_QUERY}
        skip={dontHaveLocationYet}
        variables={{ lat: this.state.lat, lon: this.state.lon }}
      >
        {({ data, loading }) => {
          if (loading || this.state.gettingPosition) {
            return <div className="Spinner" />
          }
        }}
      </Query>
    </div>
  )
}

```



And now it works, and we're reminded that the API returns Kelvin, so let's show it in Celsius and Fahrenheit (and default to the former, because it's just silly that the latter is still in use 😊):

```

const kelvinToCelsius = kelvin => Math.round(kelvin - 273.15)
const kelvinToFahrenheit = kelvin =>
  Math.round((kelvin - 273.15) * (9 / 5) + 32)

class CurrentTemperature extends Component {
  state = {
    lat: null,
    lon: null,
    gettingPosition: false,
    displayInCelsius: true
  }

  requestLocation = () => { ... }

  toggleDisplayFormat = () => {
    this.setState({
      displayInCelsius: !this.state.displayInCelsius
    })
  }

  render() {
    const dontHaveLocationYet = !this.state.lat

    return (
      <div className="Weather">
        <Query
          query={TEMPERATURE_QUERY}
          skip={dontHaveLocationYet}
          variables={{ lat: this.state.lat, lon: this.state.lon }}
        >

```

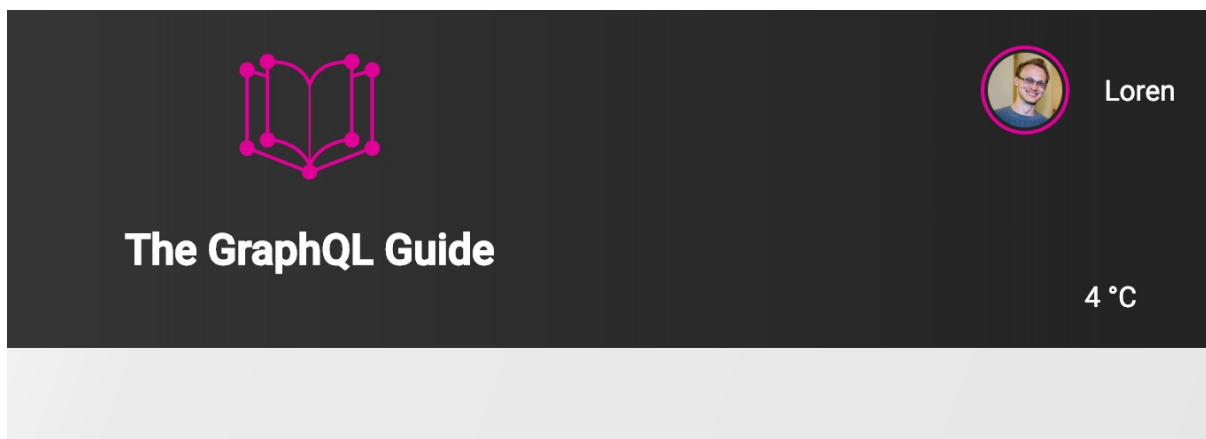
```

    ({ data, loading }) => {
      ...

      const kelvin = data.weather.main.temp
      const formattedTemp = this.state.displayInCelsius
        ? `${kelvinToCelsius(kelvin)} °C`
        : `${kelvinToFahrenheit(kelvin)} °F`

      return (
        <IconButton onClick={this.toggleDisplayFormat}>
          {formattedTemp}
        </IconButton>
      )
    }
  </Query>
</div>
)
}
}

```



To recap, we added `@rest` to our query, which made our REST link intercept the query before it was sent to our GraphQL server. The REST link returns data from the weather REST API, which gets saved to our store and provided to our component. We get all the nice things we're used to in Apollo, like declarative data fetching and loading state. And because the data is in the store, we can reuse the data in other components, and we can update the data (through requering or direct writes), and our components will automatically update.

## Review subscriptions

If you're jumping in here, `git checkout 21_0.2.0` (tag `21_0.2.0`, or compare `21...22`)

Section contents:

- [Subscription component](#)
- [Add new reviews](#)
- [Update on edit and delete](#)

Early on in this chapter we set up our [first subscription](#) for an updated GitHub star count. That was a very simple example—each event we received from the server contained a single integer:

```

type Subscription {
  githubStars: Int
}

```

In this section we'll see what it's like to work with more complex subscriptions:

```

type Subscription {

```

```

reviewCreated: Review
reviewUpdated: Review
reviewDeleted: ObjID
}

```

The first subscription sends a response event when someone creates a new review. `reviewUpdated` fires whenever a review's text or stars are edited, and `reviewDeleted` fires when one is deleted. For the first two, the events contain the review created/updated. For the last, it contains just the review's id.

## Subscription component

The first feature we'll build is a notification when the user is on the reviews page and a new review is created:

`src/components/Reviews.js`

```

import ReviewCreatedNotification from './ReviewCreatedNotification'

<main className="Reviews mui-fixed">
  ...
  <ReviewList user={user} orderBy={this.state.orderBy} />
  <ReviewCreatedNotification />
  ...
</main>

```

Now that we've got a `<ReviewCreatedNotification>` on the reviews page, what do we put in it? Apollo has a `<Subscription>` component that provides new data to its child whenever an event is received from the server:

`src/components/ReviewCreatedNotification.js`

```

import React, { Component } from 'react'
import { Subscription } from 'react-apollo'

import { ON_REVIEW_CREATED_SUBSCRIPTION } from '../graphql/Review'

class ReviewCreatedNotification extends Component {
  render() {
    return (
      <Subscription
        subscription={ON_REVIEW_CREATED_SUBSCRIPTION}
      >
        {({ data }) => console.log(data) || null}
      </Subscription>
    )
  }
}

export default ReviewCreatedNotification

```

We'll see what the event looks like in a moment, but first we need the subscription itself:

`src/graphql/Review.js`

```

export const ON_REVIEW_CREATED_SUBSCRIPTION = gql`
  subscription onReviewCreated {
    reviewCreated {
      ...ReviewEntry
    }
  }
  ${REVIEW_ENTRY}
`

```

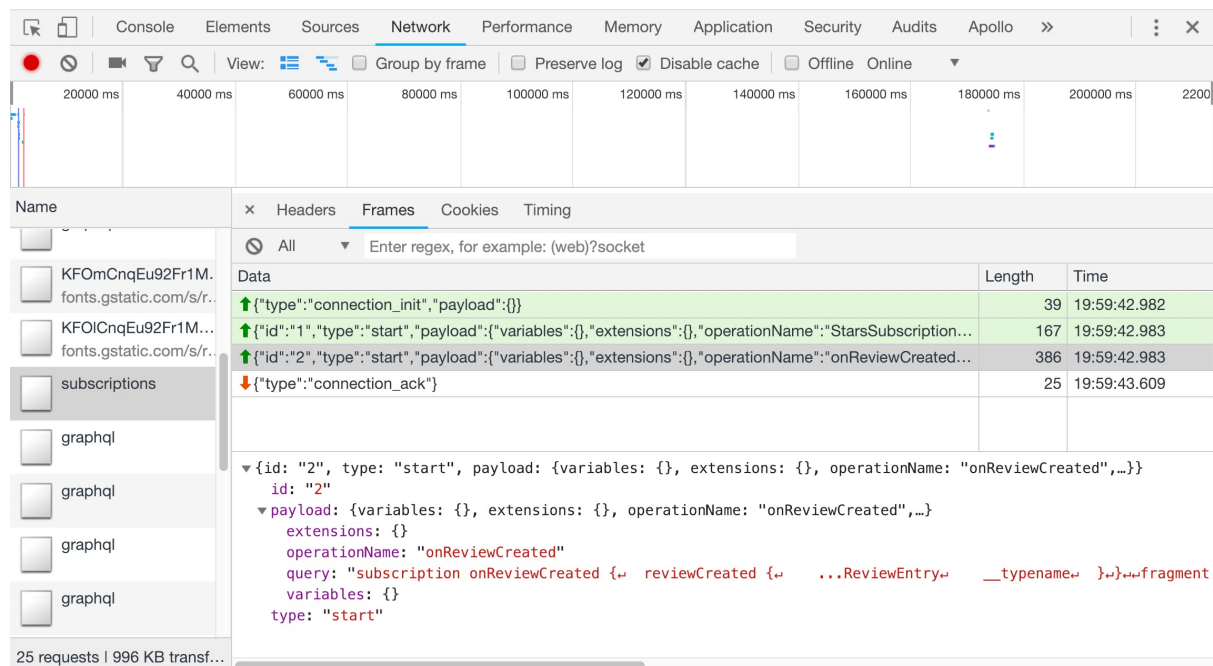
And now we can see what happens when we create a review:

- Apollo sends the `createReview` mutation to the server

- The server sends a subscription response event with data
- The `<Subscription>` component gives us the data, and we log it:

```
{
  "reviewCreated": {
    "id": "5c4b732bcd0a7103471de19b",
    "text": "It's good",
    "stars": 4,
    "createdAt": 1548448555245,
    "favorited": false,
    "author": {
      "id": "5a3cd78368e9c40096ab5e3f",
      "name": "Loren Sands-Ramshaw",
      "photo": "https://avatars2.githubusercontent.com/u/251288?v=4",
      "username": "lorensr",
      "__typename": "User"
    },
    "__typename": "Review"
  }
}
```

The data is in the same format we would expect if we made a Query named `reviewCreated`. We can also see the data arriving from the server. First let's see what it looks like initially by opening the Network tab of devtools, refreshing the page, scrolling down to "subscriptions" on the left, and selecting the "Frames" tab:



We see that the first message the client always sends once the websocket is established has `type: "connection_init"`. Then it sends two messages, each with an operation and sequential `id` numbers. They are `type: "start"` because they are starting subscriptions. The message with `id: "1"` has our GitHub stars subscription and the message with `id: "2"` has our `onReviewCreated` subscription, which we see in `payload.query`. There's also a `payload.variables` field that we're not using. If we were subscribing to a review's comments, we might use a `commentCreated(review: ObjID!): Comment` subscription, in which case we would see:

```
{
  id: "3",
  payload: {
    operationName: "onCommentCreated",
    query: "subscription onCommentCreated { commentCreated(review: $review) { id text } }",
    variables: { review: "5c4bb280cd0a7103471de19e" }
  },
  type: "start"
}
```

```
}

```

The last websocket message is from the server and has `type: "connection_ack"`, which means that the server acknowledges that it has received the `connection_init` message.

Now let's create a review and see what happens:

Data	Length	Time
↑{"type":"connection_init","payload":{}}	39	19:59:42.982
↑{"id":"1","type":"start","payload":{"variables":{},"extensions":{},"operat...	167	19:59:42.983
↑{"id":"2","type":"start","payload":{"variables":{},"extensions":{},"operat...	386	19:59:42.983
↓{"type":"connection_ack"}	25	19:59:43.609
↓{"type":"data","id":"2","payload":{"data":{"reviewCreated":{"id":"5c4b...	400	20:06:08.439

```

▼ {type: "data", id: "2", payload: {data: {,...}}}
  id: "2"
  ▼ payload: {data: {,...}}
    ▼ data: {,...}
      ▼ reviewCreated: {id: "5c4bb280cd0a7103471de19e", text: "This review will be
        ► author: {id: "5a3cd78368e9c40096ab5e3f", name: "Loren Sands-Ramshaw",...}
          createdAt: 1548464768165
          favorited: false
          id: "5c4bb280cd0a7103471de19e"
          stars: 5
          text: "This review will be sent over the websocket!"
          __typename: "Review"
        type: "data"

```

We receive another message from the server—this one with `type: "data"`, meaning it contains data! 🤖 The ID is 2, telling us that it's an event from the `onReviewCreated` subscription (which we sent to the server earlier with the matching `id: "2"`). And this time the `payload` is the same `data` object that the `<Subscription>` component gave us and we logged to the console.

But our users usually won't see messages logged to the console, so let's think about how we want to display the new review notification to the user. We could `window.alert()`, but that requires dismissal and is annoying 😡. We could put it on the page—for example in the header—but then the notification would be stuck there until either a new subscription event arrived or the page got rerendered. It doesn't need to be shown for long, taking up the user's brainspace and annoying them (at least Loren is annoyed when he can't dismiss a notification 😊). So let's show a temporary message somewhere off to the side. We can search the Material UI [component library](#) and find the component meant for this purpose—the [Snackbar](#). We control whether it's visible with an `open` prop, so we need state for that, and the `onClose` prop gets called when the user dismisses the Snackbar.

`src/components/ReviewCreatedNotification.js`

```

import Snackbar from '@material-ui/core/Snackbar'

class ReviewCreatedNotification extends Component {
  state = {
    isOpen: false
  }

  close = () => {
    this.setState({ isOpen: false })
  }

```

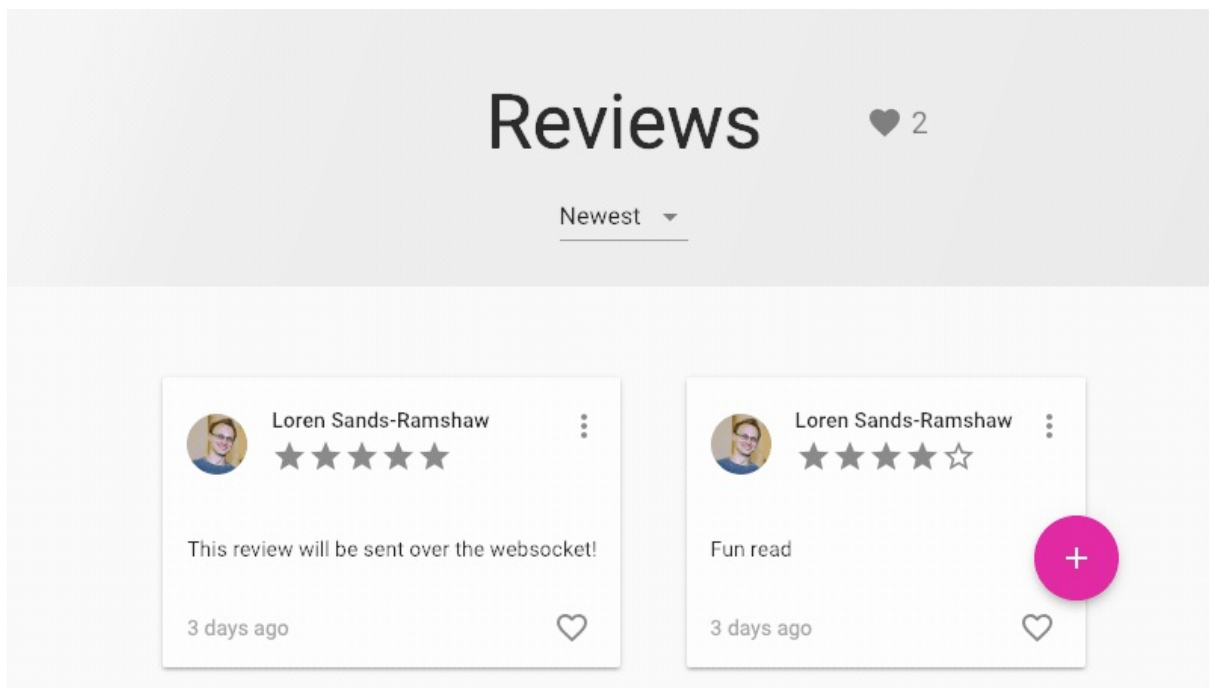
```

open = () => {
  this.setState({ isOpen: true })
  setTimeout(this.close, 5000)
}

render() {
  return (
    <Subscription
      subscription={ON_REVIEW_CREATED_SUBSCRIPTION}
      onSubscriptionData={this.open}
    >
      {({ data }) =>
        data && data.reviewCreated ? (
          <Snackbar
            anchorOrigin={{ vertical: 'bottom', horizontal: 'center' }}
            open={this.state.isOpen}
            onClose={this.close}
            message={`New review from ${data.reviewCreated.author.name}: ${
              data.reviewCreated.text
            }`}
          />
        ) : null
      }
    </Subscription>
  )
}

```

We use `isOpen` for the state. We want to set `isOpen` to true whenever we receive a new event, so we use `<Subscription>`'s `onSubscriptionData` prop. And we want to automatically dismiss the Snackbar after a few seconds, so we use a `setTimeout()`. Now when we create a review, a message slides up from the bottom of the window, stays for few seconds, and then slides back down!



gif: Review created notification

## Add new reviews

Currently when we create a review, the new review card appears at the top of the list on our page because of our optimistic update. But other users just see the notification—the review card doesn't appear in the list. Let's figure out how to get it there. One option is to turn on polling in `ReviewList.js`:

```
const withReviews = graphql(REVIEWS_QUERY, {
  options: ({ orderBy }) => ({
    pollInterval: 1000,
    errorPolicy: 'all',
    variables: { limit: 10, orderBy },
    notifyOnNetworkStatusChange: true
  }),
});
```

That's usually the easiest and best way. But this section is on subscriptions 😊. If at some point polling becomes a problem from a resource-usage perspective (usually it won't) or isn't fast enough (for example with games or chat apps), we'll want to use a subscription. We already have a subscription—the one created by our `<Subscription>` component. We could use it for two purposes. The `onSubscriptionData` prop is called with an argument of the form: `{ client, subscriptionData: { loading, data, error } }`, so we could get the new review `( subscriptionData.data.reviewCreated )` and write it to the store using `client.writeQuery()`. However, there's another way to use a subscription that's better suited to this case: the same `subscribeToMore` prop we used for `StarCount.js`. The query we want to `subscribeToMore` for is `REVIEWS_QUERY`, our list of reviews. We get the previous query result and the subscription data, and then we return a new query result:

`src/components/ReviewList.js`

```
import {
  REVIEWS_QUERY,
  REVIEW_ENTRY,
  ON_REVIEW_CREATED_SUBSCRIPTION
} from '../graphql/Review'

class ReviewList extends Component {
  componentDidMount() {
    window.addEventListener('scroll', this.handleScroll)
    this.props.subscribeToReviewUpdates()
  }
  ...
}

const withReviews = graphql(REVIEWS_QUERY, {
  options: ...,
  props: ({
    data: { reviews, fetchMore, networkStatus, subscribeToMore },
    ownProps: { orderBy }
  }) => ({
    reviews,
    networkStatus,
    loadMoreReviews: ...,
    subscribeToReviewUpdates: () => {
      subscribeToMore({
        document: ON_REVIEW_CREATED_SUBSCRIPTION,
        updateQuery: (prev, { subscriptionData }) => {
          // Assuming infinite reviews, we don't need to add new reviews to
          // Oldest list
          if (orderBy === 'createdAt_ASC') {
            return prev
          }

          const newReview = subscriptionData.data.reviewCreated
          return {
            reviews: [newReview, ...prev.reviews]
          }
        }
      })
    }
  })
})
```

Now when we're viewing the most recent reviews ( `createdAt_DESC` ) and receive a subscription event, we add the new review to the front of the list of reviews, and it appears first on the page. We can test this out by opening a second browser tab, creating a new review in that tab, and seeing it immediately appear in the first tab.



## Update on edit and delete

It would also be nice to update the reviews when someone else edits or deletes them. If we look at the Playground schema, we can see that the server has more subscription options related to reviews: `reviewUpdated: Review` and `reviewDeleted: objID`. So let's use 'em! Step 1 is writing the subscription documents and step 2 is adding more calls to `subscribeToMore`. (`subscribeToMore` doesn't mean that we're necessarily subscribing to new documents—just that we're subscribing to more related data, and in this case the data is either the review that was updated or the ID of the review that was deleted.) First, the documents:

`src/graphql/Review.js`

```
export const ON_REVIEW_UPDATED_SUBSCRIPTION = gql`
  subscription onReviewUpdated {
    reviewUpdated {
      ...ReviewEntry
    }
  }
  ${REVIEW_ENTRY}
`

export const ON_REVIEW_DELETED_SUBSCRIPTION = gql`
  subscription onReviewDeleted {
    reviewDeleted
  }
`
```

Because the return type of `reviewDeleted` is a scalar (a custom one called `objID`), we don't write a selection set. `subscriptionData.data.reviewDeleted` will be an `objID` string, not an object. Next, `subscribeToMore`:

`src/components/ReviewList.js`

```
import reject from 'lodash/reject'

import {
  REVIEWS_QUERY,
  REVIEW_ENTRY,
  ON_REVIEW_CREATED_SUBSCRIPTION,
  ON_REVIEW_UPDATED_SUBSCRIPTION,
  ON_REVIEW_DELETED_SUBSCRIPTION
} from '../graphql/Review'

...

const withReviews = graphql(REVIEWS_QUERY, {
  options: ...,
  props: ({
    data: { reviews, fetchMore, networkStatus, subscribeToMore },
    ownProps: { orderBy }
  }) => ({
    reviews,
    networkStatus,
    loadMoreReviews: ...,
    subscribeToReviewUpdates: () => {
      subscribeToMore({
        document: ON_REVIEW_CREATED_SUBSCRIPTION,
        updateQuery: ...
      })
      subscribeToMore({
        document: ON_REVIEW_UPDATED_SUBSCRIPTION,
        updateQuery: (prev, { subscriptionData }) => {
          const updatedReview = subscriptionData.data.reviewUpdated
          return {
            reviews: prev.reviews.map(review =>
              review.id === updatedReview.id ? updatedReview : review
            )
          }
        }
      })
    }
  })
})
```

```

    subscribeToMore({
      document: ON_REVIEW_DELETED_SUBSCRIPTION,
      updateQuery: (prev, { subscriptionData }) => {
        const deletedId = subscriptionData.data.reviewDeleted
        return {
          reviews: reject(prev.reviews, { id: deletedId })
        }
      }
    })
  })
}
})
})
})

```

For review updates, we replace the review in the list from the store ( `prev` ) with the updated one we get from the subscription. For deletions, we remove it from the list.

## Prefetching

Background: [browser performance](#)

Section contents:

- [On mouseover](#)
- [Cache redirects](#)

Prefetching is fetching data from the server before we need it so that when we do need it, we already have it on the client and can use it right away. This is great for UX because the user doesn't have to look at a loading screen waiting for data to load. It's a common pattern—both [Gatsby](#) and [Next.js](#) prefetch entire webpages with their `<Link>` components.

The most useful thing to prefetch in our app is the section content! We can prefetch just by making a query with the Apollo client:

```

client.query({
  query: ...
})

```

This will place the results in the store, so that when we render a `<Section>` and it makes a query for section data, it will immediately find the data in the store. We could prefetch all the sections using the `sections` root query:

```

import { withApollo } from 'react-apollo'

class App extends Component {
  componentDidMount() {
    requestIdleCallback(() => {
      this.props.client.query({
        query: ALL_SECTIONS
      })
    })
  }
}

render() { ... }
}

const ALL_SECTIONS = gql`
  query AllSections {
    sections {
      id
      content
      views
    }
  }
`

```

```
export default withApollo(App)
```

For the query selection set, we check the queries in `Section.js` and see that it needs the `content` and `views`. We use `withApollo` to get access to the `client`, and we use `requestIdleCallback()` (which calls the callback when the browser isn't busy) so that we don't delay any of the work involved with the initial app render. When the `AllSections` query response arrives, the data is put in the store, and any future render of `<Section>` is immediate, without need to talk to the server.

## On mouseover

If you're jumping in here, `git checkout 22_0.2.0` (tag [22\\_0.2.0](#), or compare [22...23](#))

The potential issue with the above approach is how much data we're prefetching—the entire content of the book. The more data we fetch, the more work the server has to do, and the more work the client has to do—first to receive and store it, and then later to interact with the larger store. The client's workload is more likely to become an issue because Apollo runs in the main thread (it interacts with React, which interacts with the DOM, which is in the main thread), and things it does might delay user interaction or freeze animations (see [Background > Browser performance](#) for more info). It takes longer for Apollo to query and update the store when there's more data in the store.

So usually instead of prefetching all of the data we could possibly need, we selectively prefetch some of it. One common way to do this is prefetching when the user mouses over something clickable. We might know that we'll need certain data if they click that particular link or button, in which case we can fetch the data when the mouseover happens instead of waiting for the click. It's possible that they won't click, in which case we'll have extra data that we don't need, but this usually isn't a problem.

For the Guide, when a user hovers over a link in the table of contents, we know what data we'll need—that section's contents. We can export the query for section contents from `Section.js` and use it in `TableOfContents.js` to make the query. In order to make the query, we need access to the client instance, so we use `withApollo()`:

`src/components/TableOfContents.js`

```
import { graphql, withApollo, compose } from 'react-apollo'
import { ApolloClient } from 'apollo-client'

TableOfContents.propTypes = {
  ...
  client: PropTypes.instanceOf(ApolloClient).isRequired
}

export default compose(
  withData,
  withApollo
)(TableOfContents)
```

Now we can make the query inside the `onMouseOver` function:

```
import { SECTION_BY_ID_QUERY } from './Section'

const TableOfContents = ({ chapters, loading, client }) => (
  <nav className="TableOfContents">
    ...
    <NavLink
      to={{
        pathname: slugify(chapter),
        state: { chapter, section: chapter.sections[0] }
      }}
      className="TableOfContents-chapter-link"
      activeClassName="active"
      isActive={(match, location) => {
        const rootPath = location.pathname.split('/')[1]
        return rootPath.includes(withHyphens(chapter.title))
      }}
    >
      {chapter.title}
    </NavLink>
  </nav>
)
```

```

    }}
    onMouseOver={() => {
      client.query({
        query: SECTION_BY_ID_QUERY,
        variables: {
          id: chapter.sections[0].id
        }
      })
    }}
  }}
  >
  ...
  <NavLink
    to={{
      pathname: slugify(chapter, section),
      state: { chapter, section }
    }}
    className="TableOfContents-section-link"
    activeClassName="active"
    onMouseOver={() => {
      client.query({
        query: SECTION_BY_ID_QUERY,
        variables: {
          id: section.id
        }
      })
    }}
  >
  </nav>
)

```

We have two `onMouseOver` s: When mousing over a chapter link, we query for the first section of that chapter. When mousing over a section link, we query for that section.

We also need to add the export:

`src/components/Section.js`

```

export const SECTION_BY_ID_QUERY = gql`
  query SectionContent($id: String!) {
    section(id: $id) {
      id
      content
      views
      scrollY @client
    }
  }
`

```

And now it works! When the user clicks the link, `<Section>` makes a new query, but instantly renders the section content because it's already in the store. We can check this in two ways:

- Opening the devtools Network tab and watching when the `SectionContent` query is sent to the server.
- Seeing whether the loading skeleton appears when we hover over a new link for a second before clicking, versus immediately clicking it. If we want to see the difference more clearly, we can slow down the connection to “Fast 3G” in the devtools Network tab.

Depending on how long we hover, we may still see the loading skeleton: for example, if it takes three seconds to load when we immediately click, and then we hover on the next link for two seconds before clicking, we will still see the skeleton for one second.

One issue to consider is whether we're making a lot of extra queries, because users may mouse over sections that we've already loaded. But the default Apollo client network policy is `cache-first`, which means that if Apollo finds the query results in the cache, it won't send the query to the server. We're using the default, so we don't need to do anything, but if we had set a different default in the `ApolloClient` constructor like this:

`src/lib/apollo.js`

```
export const apollo = new ApolloClient({
  link,
  cache,
  defaultOptions: {
    query: {
      fetchPolicy: 'cache-and-network'
    }
  }
})
```

`cache-and-network` immediately returns any results available in the cache *and* also queries the server

then we could set a different network policy just for our prefetching:

```
onMouseOver={() => {
  client.query({
    query: SECTION_BY_ID_QUERY,
    variables: {
      id: section.id
    },
    fetchPolicy: 'cache-first'
  })
}}
```

## Cache redirects

If you're jumping in here, `git checkout 23_0.2.0` (tag [23\\_0.2.0](#), or compare [23...24](#))

There are often more ways than just mouseovers to intelligently prefetch certain data. What the ways are depends on the type of app. We have to think about how the user uses the app, and what they might do next. In our app, one common action will probably be to read the next section. So a simple thing we can do is whenever we show a section, we prefetch the next section. Let's first get the `client` instance into `<Section>`. Before we used `withApollo()`, which we could do again, but we don't need to—our `<Section>` is inside a `<Query>`, which always provides a client instance:

`src/components/Section.js`

```
import { ApolloClient } from 'apollo-client'

class Section extends Component {
  ...
}

Section.propTypes = {
  ...
  client: PropTypes.instanceOf(ApolloClient).isRequired
}

const SectionWithData =
  ...
  <Query query={query} variables={variables}>
    {queryInfo => (
      ...
      <Section
        {...createProps(queryInfo)}
        client={queryInfo.client}
      />
    )}
  </Query>
```

Now inside `Section`, we want to make the query on initial render and when the section changes. We're currently repeating things inside `componentDidMount()` and `componentDidUpdate()`, so let's refactor out a new method `onSectionChange()`:

```

class Section extends Component {
  onSectionChange = newId => {
    this.viewedSection(newId)
    this.updateScrollPosition()
  }

  ...

  componentDidMount() {
    window.addEventListener('scroll', this.handleScroll)

    if (this.props.section) {
      this.onSectionChange(this.props.section.id)
    }
  }

  ...

  componentDidUpdate(prevProps) {
    if (!this.props.section) {
      return
    }

    const { id } = this.props.section
    const sectionChanged = get(prevProps, 'section.id') !== id

    if (sectionChanged) {
      this.onSectionChange(id)
    }
  }
  ...
}

```

And now inside `onSectionChange()` we can do the prefetching:

```

class Section extends Component {
  onSectionChange = newId => {
    this.viewedSection(newId)
    this.updateScrollPosition()
    this.prefetchNextSection(newId)
  }

  prefetchNextSection = currentId => {
    this.props.client.query({
      query: ...
      variables: {
        id: ...
      }
    })
  }

  ...
}

```

But what query do we make? We could take the current section ID, eg `1_3` (chapter 1, section 3) and try the next section number, eg `1-4`, and if that failed (because it was the end of the chapter), we could go to the next chapter with `2_1`. That would look something like:

```

prefetchNextSection = async currentId => {
  const nextSectionId = ...
  const { data } = await this.props.client.query({
    query: SECTION_BY_ID_QUERY,
    variables: {
      id: nextSectionId
    }
  })

  if (!data.section) {
    const nextChapterId = ...
  }
}

```

```

    this.props.client.query({
      query: SECTION_BY_ID_QUERY,
      variables: {
        id: nextChapterId
      }
    })
  }
}

```

`client.query()` returns a Promise, which we can `await`, and our API resolves the `section` query to `null` when there is no such section. So when `data.section` is null, we query for the next chapter. (Alternatively, if our API instead returned a “No such section” error, we could use a `try...catch` statement.)

However, there's a way to get the next section in a single query—the `Section` type has a field `next` of type `Section`! Let's write a query for that:

```

prefetchNextSection = currentId => {
  this.props.client.query({
    query: NEXT_SECTION_QUERY,
    variables: {
      id: currentId
    }
  })
}
...

const NEXT_SECTION_QUERY = gql`
  query NextSection($id: String!) {
    section(id: $id) {
      id
      next {
        id
        content
        views
        scrolly @client
      }
    }
  }
`

```

For the `next` selection set, we copy the fields from the other queries in `Section.js`, since those are the fields that will be needed if the user navigates to the next section. It now seems like we're done, and if we look at the Network tab, we see that the `prefetch` query is made. We can also see in Apollo devtools that the `Section` object with the next section ID is in the store. However, when we navigate to the next section, the `SectionContent` query is being made!

```

query SectionContent($id: String!) {
  section(id: $id) {
    id
    content
    views
    scrolly @client
  }
}

```

The problem is that Apollo doesn't have a way of knowing that the server will respond to a `section` query that has an `id` argument with the `Section` object matching that ID. We can inform Apollo of this using a `cache redirect`, which is a configuration function we provide to `InMemoryCache` that returns the cache key Apollo should look for before it sends the query to the server. If it finds an object in the store under that key, it will just return that.

`src/lib/apollo.js`

```

const cache = new InMemoryCache({
  cacheRedirects: {

```

```

Query: {
  section: (_, { id }, { getCacheKey }) =>
    getCacheKey({ __typename: 'Section', id })
}
}
})

```

`getCacheKey()` formats the object into the cache key—we’re using the default cache key format, `__typename:id`, for example `Section:1-4`, which is what we see in the Cache tab of Apollo devtools.

Now our prefetching works. If we turn on Slow 3G in the Network tab and click on the next section, it will render immediately, because it was prefetched when the previous section rendered 😊.

## Batching

If you’re jumping in here, `git checkout 24_0.2.0` (tag [24\\_0.2.0](#)). We won’t be leaving the code from this section in our app, so the next section will also start at tag `24`.

If we load the site with the Network tab of devtools open, we see a lot of requests that say “graphql” on the left—that’s the path, so the full endpoint is `api.graphql.guide/graphql`, our GraphQL API. By default, each of the GraphQL queries in our app is sent in its own HTTP request. We can look at the request payload to see which query it is, for example our simple `starsQuery`:

Name	Headers	Preview	Response	Timing
subscriptions api.graphql.guide				
graphql api.graphql.guide				
graphql api.graphql.guide				
graphql api.graphql.guide				
graphql api.graphql.guide				
info?t=154889694221 /sockjs-node				
graphql devtools.apollodata...				

**General**

**Request URL:** https://api.graphql.guide/graphql

**Request Method:** POST

**Status Code:** 200

**Remote Address:** 104.27.190.39:443

**Referrer Policy:** no-referrer-when-downgrade

**Response Headers (16)**

**Request Headers (6)**

**Request Payload** [view source](#)

```

{operationName: "StarsQuery", variables: {}, query: "query StarsQuery { githubStars }"}
operationName: "StarsQuery"
query: "query StarsQuery { githubStars }"
variables: {}

```

We can **batch** our initial queries into one request, which will look like this:



**Name**

- logo.9fb731ea.svg /static/media
- subscriptions api.graphql.guide
- graphql api.graphql.guide
- info?t=154889980275 /sockjs-node
- graphql devtools.apollodata...
- websocket /sockjs-node/193/5u...
- favicon.ico
- graphql devtools.apollodata...
- graphql api.graphql.guide

**Headers** Preview Response Timing

**General**

**Request URL:** https://api.graphql.guide/graphql

**Request Method:** POST

**Status Code:** 200

**Remote Address:** 104.27.191.39:443

**Referrer Policy:** no-referrer-when-downgrade

**Response Headers (15)**

**Request Headers (6)**

**Request Payload** view source

```
[{operationName: "StarsQuery", variables: {}, query: "query StarsQuery {
  stars {
    id
    title
    description
    createdAt
    updatedAt
    githubStars
  }
}"}]
```

We also see that the third request is to `/graphql`, but the Request Method is `OPTIONS` instead of the normal `POST`, and the status code is `204` instead of the normal `200`. This is called a **preflight** request that Chrome makes to the server to check its security policy ([CORS](#)), since it's going to a different domain from the client (`localhost:3000`).

At first glance, it seems better to batch—fewer requests is more efficient for our browser, and it reduces the HTTP request load on our server. However, the big drawback is that we only get one response. This means that the server keeps all of our results until the last query in the batch completes, and *then* sends all the results back to us together in one response. Without batching, we get results to our faster queries faster, and those parts of the page get rendered, while the other parts stay in loading state for longer. For this reason, it's recommended that we keep the default unbatched requests, and only try batching when we have server load issues *and* have [already made other performance improvements](#). If we ever get to that point, here's the simple setup:

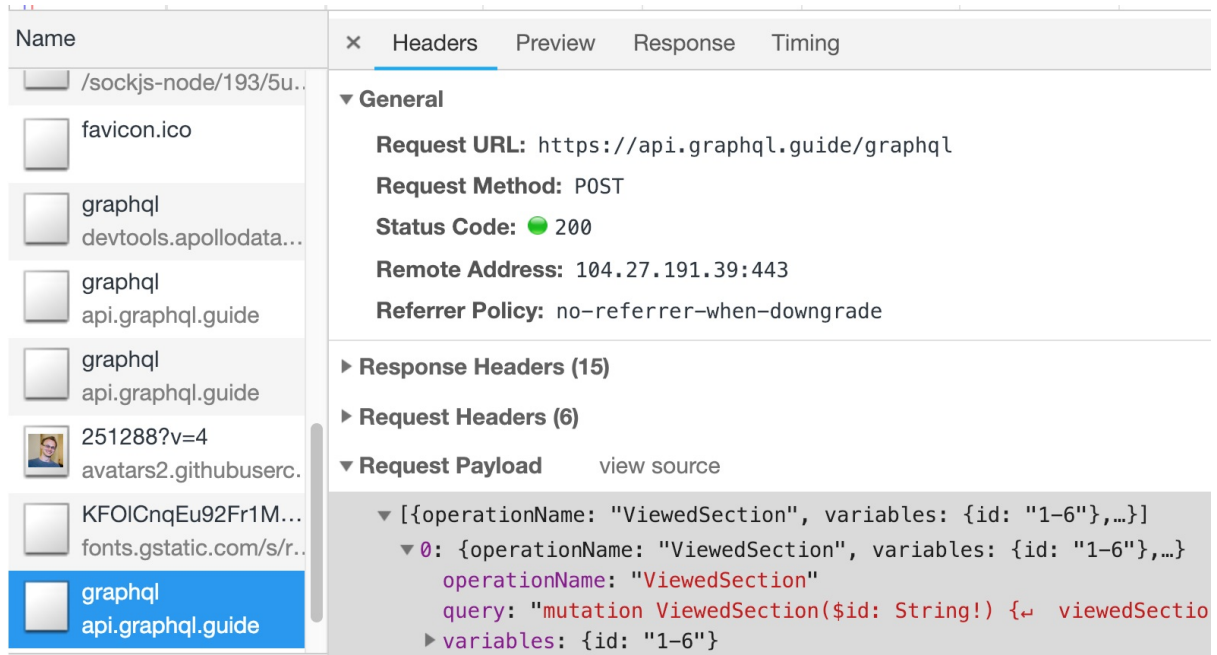
```
npm install apollo-link-batch-http
```

```
src/lib/apollo.js
```

```
import { BatchHttpLink } from 'apollo-link-batch-http'

const httpLink = new BatchHttpLink({ uri: 'https://api.graphql.guide/graphql' })
```

We replace our previous `httpLink` with the link from `apollo-link-batch-http`. One thing you may notice in the Network tab is that soon after our initial batched request, we see another—this one only contains a single operation, named `ViewedSection`:



The reason this wasn't included in the initial batch request is because it happens a second later: only queries that are made within a certain window are batched together. The default `batchInterval` is 10 milliseconds, and can be changed [as an option](#) to `BatchHttpLink()`.

If we know there are certain queries that will take longer than others, and we want them to bypass batching, we can set up both a normal http link and a batched link. Then we can use `split()` to decide which link to send a request to:

```
const client = new ApolloClient({
  link: split(
    operation => operation.getContext().slow === true,
    httpLink,
    batchHttpLink
  )
});

<Query query={SLOW_QUERY} context={{ slow: true }}>
<Query query={NORMAL_QUERY}>
```

We add data to the context, and then we check it inside `split()`: if the context has `slow: true`, then send via the `httpLink`. Otherwise, send via the `batchHttpLink`.

## Persisting

If you're jumping in here, `git checkout 24_0.2.0` (tag [24\\_0.2.0](#), or compare [24...25](#))

The Apollo store is stored in page-specific memory. When the webpage is closed or reloaded, the memory is cleared, which means the next time our app loads, the store is empty—it has to fetch all the data it needs from the server again. **Persisting** is saving the data in the Apollo store so that on future pageloads, we can restore the data to the store, and we don't have to fetch it. The main benefit is we can show the data to the user much faster than we could if we had to fetch it from the server. We can easily set this up with the `apollo-cache-persist` package:

`src/components/App.js`

```
import { persistCache } from 'apollo-cache-persist'

import { cache } from '../lib/apollo'
```

```
persistCache({
  cache,
  storage: window.localStorage,
  maxSize: 4500000, // little less than 5 MB
  debug: true
})
```

And we need to export `cache` :

```
src/lib/apollo.js
```

```
export const cache = new InMemoryCache({ ... })
```

The `persistCache()` function sets up persistence. `debug: true` has it log the size of the cache whenever it's saved. The `storage` parameter has a number of options:

- `window.localStorage`
- `window.sessionStorage`
- `localForage`: uses WebSQL or IndexedDB when available (most browsers), and falls back to `localStorage`
- `AsyncStorage` in React Native

`sessionStorage` is rarely used, since it is cleared when the browser is closed, and we usually want to store data for a longer period. `localStorage` is simple to use and can consistently store 5–10 MB. `localForage` is good for complex querying and larger sets of data. However, it is generally slower than `localStorage` for simple operations (and our operation is simple: it's just saving and getting a single piece of data—the contents of the Apollo store). We also have to import it from npm, which adds an additional 8 KB gzipped to our JavaScript bundle.

So we probably would only want to use `localForage` if we needed more than 5 MB of space. Let's think about what kind of data our app queries for, how much space it takes up, and how much we might want of it. The largest thing the Guide queries for is section text, and according our new logging, each section (currently just a paragraph of Lorem ipsum) takes up 2 KB:

```
[apollo-cache-persist] Persisted cache of size 34902
[apollo-cache-persist] Persisted cache of size 37014
```

The second line was printed after hovering over a section link in the table of contents.

At this rate, we would fill up the cache after loading  $5000 \text{ KB} / 2 \text{ KB} = 2500$  sections, so 5 MB is currently plenty of room for us. Let's go with `localStorage`.

`maxSize` is the maximum number of bytes to persist. When `maxSize` is reached, it will stop saving data changes in the current session, and the next time the app starts, the cache will be cleared. We could set a different `maxSize` depending on which browser we're in, but for simplicity let's just assume we're in the [lowest-quota browser](#), Safari, which can store 5 MB. We set `maxSize` to 4.5 MB to leave a little room for other uses (for instance our Auth0 library uses `localStorage`, and maybe we'll decide later that we want to use it for something else).

Alright—we've covered all of the arguments we used with `persistCache()` (there are others we're not using). But we're not done: the cache is getting persisted fine, but when a saved cache is restored on subsequent pageloads, our components are still querying, and they don't get data until the query response comes back from the server.

We can verify this by changing the speed to "Slow 3G" in Network devtools and see A) the graphql requests being sent and B) both the first load and subsequent loads take a few seconds for the loading skeleton to be replaced with text.

The reason for this is that `persistCache()` takes time to complete (at least 150 ms on Loren's computer), and by that time, `react-apollo` has already sent off our components' queries. And when it does complete, our components don't know that there's new data in the cache. So when there's a saved cache to restore, we want to wait for

`persistCache()` to complete before rendering our components and triggering their queries. Then all of our `cache-first` queries will see that the data is in the cache and use it instead of requesting it from the server. We can tell if there's a saved cache by checking in `localStorage` for the key that `persistCache()` uses, `apollo-cache-persist` :

`src/components/App.js`

```
const cacheHasBeenSaved = !!localStorage.getItem('apollo-cache-persist')

class App extends Component {
  state = {
    loading: cacheHasBeenSaved
  }

  async componentDidMount() {
    await persistCache({
      cache,
      storage: window.localStorage,
      maxSize: 4500000, // little less than 5 MB
      debug: true
    })

    this.setState({
      loading: false
    })
  }

  render() {
    if (this.state.loading) {
      return null
    }

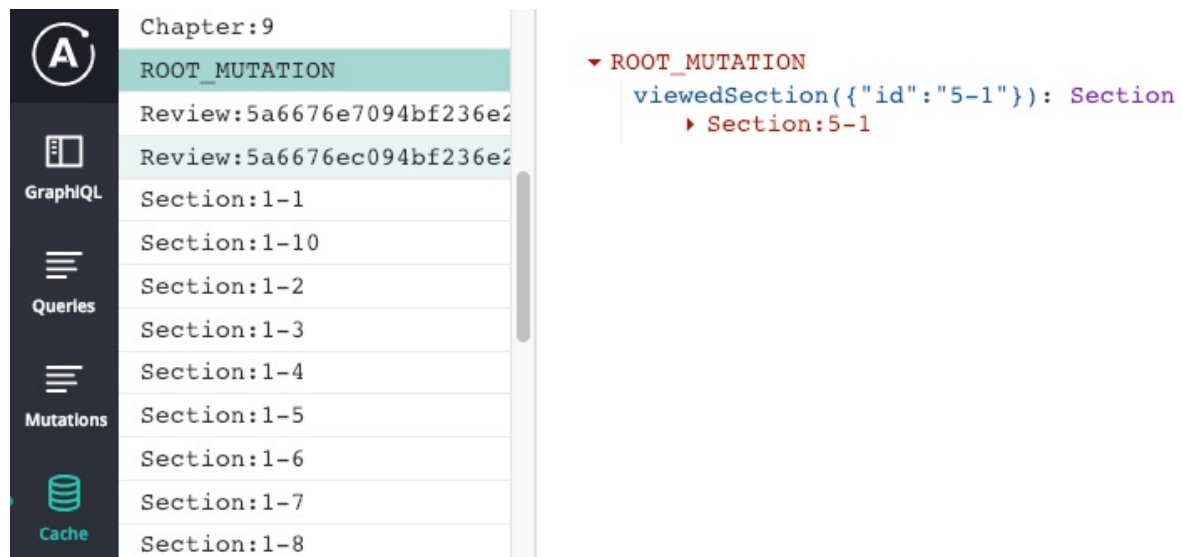
    return (
      <div className="App">
        ...
      </div>
    )
  }
}

export default App
```

Now let's test it out. When we load the app for the first time, we see something like this:

```
[apollo-cache-persist] No stored cache to restore
[apollo-cache-persist] Persisted cache of size 17005
[apollo-cache-persist] Persisted cache of size 17129
```

The first message prints out on load, and the second appears a second after the page content appears, saying that the Apollo cache was saved to `localStorage` and what its size was. The third appears shortly after that, meaning the cache was re-saved, and the size only goes up by about a hundred bytes. What caused the re-save? We must have made another request to the server after the initial set of requests. We can check the Network tab to see what the last GraphQL request was, and we see that it's the `ViewedSection` mutation. But why would that mutation change the Apollo cache? It's not a query fetching data. Let's look at the cache to see. In the Cache tab of Apollo devtools, there's a `ROOT_MUTATION` :



We see that our mutation is indeed in the cache, and it resolved to a `Section` object. Is the entire cache, including mutation results, persisted? We can look at what's saved by entering this in the browser console:

```
JSON.parse(localStorage.getItem('apollo-cache-persist'))
```

```

▶ Chapter:-2: {id: -2, number: null, title:
▼ ROOT_MUTATION:
  ▼ viewedSection({"id":"5-1"}):
    generated: false
    id: "Section:5-1"
    type: "id"
    typename: "Section"
    ▶ __proto__: Object
  ▶ __proto__: Object
▶ ROOT_QUERY: {loginInProgress: false, githu
▶ Review:5a6676e7094bf236e215f487: {id: "5a6

```

And we see that it is present, and the `viewedSection` mutation has `type: "id"`, meaning that it has been normalized, linking to the top-level object with `id: "Section:5-1"`.

Now let's see what happens when we reload the app.

```

[apollo-cache-persist] Restored cache of size 17129
[apollo-cache-persist] Persisted cache of size 17129

```

The cache is restored! We can check to make sure the cache is being used to immediately provide data to our components by: 1) seeing in Network devtools that our initial batch of GraphQL requests are not being made, and 2) slowing the network speed to "Slow 3G" and seeing that there is no loading skeleton. Versus if we delete the cache and reload, we see the skeleton for a few seconds:

- Application devtools
- Select `Local Storage` on the left
- Select `http://localhost:3000`
- Select `apollo-cache-persist` on the right
- Click the `x` delete button
- Reload

So the persisting is working correctly, but if we test the app further, we find that we can't log out! Well, technically, we can, but it doesn't look like we are—after clicking “Sign out” on the profile page, the site reloads and we still see our GitHub profile photo on the top-right, and we can still click it to see our profile. Why is that?

On load, the app reads all the queries from the cache, including the `currentUser` query, which was saved to the cache when we logged in. It's still there, along with all the private data we had access to, like bonus chapters. To fix this, we can clear the cache when we log out. In order to clear the cache, we need to use a different API from `apollo-cache-persist`. We've been using the basic API, `persistCache()`. The more advanced API is `CachePersistor`:

```
const persistor = new CachePersistor(options)
```

And then we call methods on the `persistor` object when we want things to happen: for instance, `persistor.restore()` when we want to restore the cache (which `persistCache()` did automatically, but now we need to do ourselves). So let's update `App.js`:

`src/components/App.js`

```
import { CachePersistor } from 'apollo-cache-persist'

import { cache, apollo } from '../lib/apollo'

const persistor = new CachePersistor({
  cache,
  storage: window.localStorage,
  maxSize: 4500000, // little less than 5 MB
  debug: true
})

apollo.onResetStore(() => persistor.purge())

const cacheHasBeenSaved = !!localStorage.getItem('apollo-cache-persist')

class App extends Component {
  state = {
    loading: cacheHasBeenSaved
  }

  async componentDidMount() {
    await persistor.restore()

    this.setState({
      loading: false
    })
  }

  ...
}
```

This line clears the cache when the store is reset:

```
apollo.onResetStore(() => persistor.purge())
```

And since we call `apollo.resetStore()` on logout in `src/lib/auth.js`, clicking “Sign out” clears the cache, and we see “Sign in” instead of our photo! ✅

But there's another bug! 😞 When we're signed out, we get truncated section content back from the API. This gets saved in the cache, and when we sign in, the current section gets refetched (due to `apollo.refetchObservableQueries()` being called in `auth.js` on login). But if we looked at more than the current section before signing in, the other sections don't get refetched, because there are no current (“observable”) queries for them. So they get stuck with the truncated content—when we revisit them, the truncated content is loaded from the cache. We can make sure they're updated either by:

- changing the section content queries' fetch policy to `cache-and-network`, or
- replacing `apollo.refetchObservableQueries()` with `apollo.resetStore()`

The second would be simpler, but let's do the first, because it also fixes another issue: when data is cached, it's saved until it reaches `maxSize`, which could take a long time. The book content will periodically be updated, and we want our users to see the updated content. With `cache-and-network`, the latest version will always be fetched from the server. We make the change by adding the `fetchPolicy` prop to our `<Query>` component:

`src/components/Section.js`

```
const SectionWithData = ({ location: { state, pathname } }) => {
  ...
  return (
    <Query query={query} variables={variables} fetchPolicy="cache-and-network">
```

And we can test with these steps:

- Sign out
- Click “Preface” and then “Introduction”
- Sign in
- Click “Preface”

The preface content is no longer truncated, but we see a loading skeleton before the full content appears. So `loading` must be initially true, even though we have the truncated preface content in the cache. This is because `loading` is true whenever there is a network request in progress (which there is, because we're using `cache-and-network`). And we see the skeleton when loading any section—even those with full content in the cache. It's as if we don't even have a store anymore. To stop showing the skeleton, we have to go by whether there's data instead of using Apollo's `loading` variable. So let's set `loading: x` in each `createProps` function:

```
const SectionWithData = ({ location: { state, pathname } }) => {
  const page = deslugify(pathname)

  let query, variables, createProps

  if (state) {
    query = SECTION_BY_ID_QUERY
    variables = { id: state.section.id }
    createProps = ({ data }) => ({
      section: {
        ...state.section,
        content: get(data, 'section.content'),
        views: get(data, 'section.views'),
        scrollY: get(data, 'section.scrollY')
      },
      chapter: state.chapter,
      loading: !data.section
    })
  } else if (page.chapterTitle) {
    query = SECTION_BY_CHAPTER_TITLE_QUERY
    variables = { title: page.chapterTitle }
    createProps = ({ data }) => ({
      section: get(data, 'chapterByTitle.section'),
      chapter: {
        ...data.chapterByTitle,
        number: null
      },
      loading: !data.chapterByTitle
    })
  } else if (page.chapterNumber) {
    query = SECTION_BY_NUMBER_QUERY
    variables = page
    createProps = ({ data }) => ({
      section: get(data, 'chapterByNumber.section'),
      chapter: data.chapterByNumber,
      loading: !data.chapterByNumber
    })
  }
```

```

    })
  }

  return (
    <Query query={query} variables={variables} fetchPolicy="cache-and-network">
      {queryInfo => (
        <Mutation mutation={VIEWED_SECTION_MUTATION}>
          {viewedSection => (
            <Mutation mutation={SET_SECTION_SCROLL_MUTATION}>
              {setScrollPosition => (
                <Section
                  {...createProps(queryInfo)}
                  client={queryInfo.client}
                  viewedSection={viewedSection}
                  setScrollPosition={setScrollPosition}
                />
              )}
            </Mutation>
          )}
        </Mutation>
      )}
    </Query>
  )
}

```

And now it works! When we revisit the preface, it shows the truncated content in the cache first, and then shows the full content fetched from the server.

While the problem we were trying to fix is fixed, the astute will notice [a new bug](#), which is pending a fix in `react-apollo`.

## Multiple endpoints

If you're jumping in here, `git checkout 25_0.2.0` (tag [25\\_0.2.0](#), or compare [25...26](#))

So far we've been working with a single GraphQL endpoint, `api.graphql.guide/graphql` (and its websocket counterpart, `/subscriptions`). Would we ever want our app to talk to another endpoint? Maybe. Similarly to the APIs in the [REST section](#), we usually would want to proxy the other GraphQL endpoint through our GraphQL server (we'll go over how to do this in the server chapter). There are two main reasons:

- If the endpoint is authenticated, we usually will want to keep it private on our server.
- It's nice for our GraphQL endpoint to have the complete graph of data our app might need, so that devs have one source of truth, and so that our server-side tools—including caching, logging, and analytics—cover all of our queries.

However, there are cases in which we might not want to proxy: we might not have control over the backend, or maybe we want to reduce load on our server or get a slightly better latency than we would while proxying. So we need a GraphQL API from which to fetch some data for this section. Apollo GraphQL shares the name of NASA's Apollo project, which landed the first humans on the moon in 1969. And Apollo GraphQL identifies with the rocket emoji 🚀. So let's put that emoji somewhere and make it an easter egg—if it's clicked, we'll show the next SpaceX launch using the unofficial [SpaceX GraphQL API](#).

So far, all of our queries know what endpoint to talk to because of the `<ApolloProvider>` wrapped around the `<App>`:

`src/index.js`

```

ReactDOM.render(
  <BrowserRouter>
    <ApolloProvider client={apollo}>
      <MuiThemeProvider theme={theme}>
        <App />
      </MuiThemeProvider>
    </ApolloProvider>
  )

```



```

</BrowserRouter>,
document.getElementById('root')
)

```

where `apollo` is the `ApolloClient` instance we created with an HTTP link to `api.graphql.guide/graphql` :

`src/lib/apollo.js`

```

const httpLink = createHttpLink({
  uri: 'https://api.graphql.guide/graphql'
})

...

export const apollo = new ApolloClient({ link, cache })

```

We're going to need a second `ApolloClient` instance to use for our launch query:

`src/lib/apollo.js`

```

export const apolloSpace = new ApolloClient({
  link: ApolloLink.from([
    errorLink,
    createHttpLink({
      uri: 'https://api.spacex.land/graphql'
    })
  ]),
  cache: new InMemoryCache()
})

```

Now to use it, we can put it in the `client` prop of `<Query>`, which overrides its normal behavior of using the client provided by `<ApolloProvider>`.

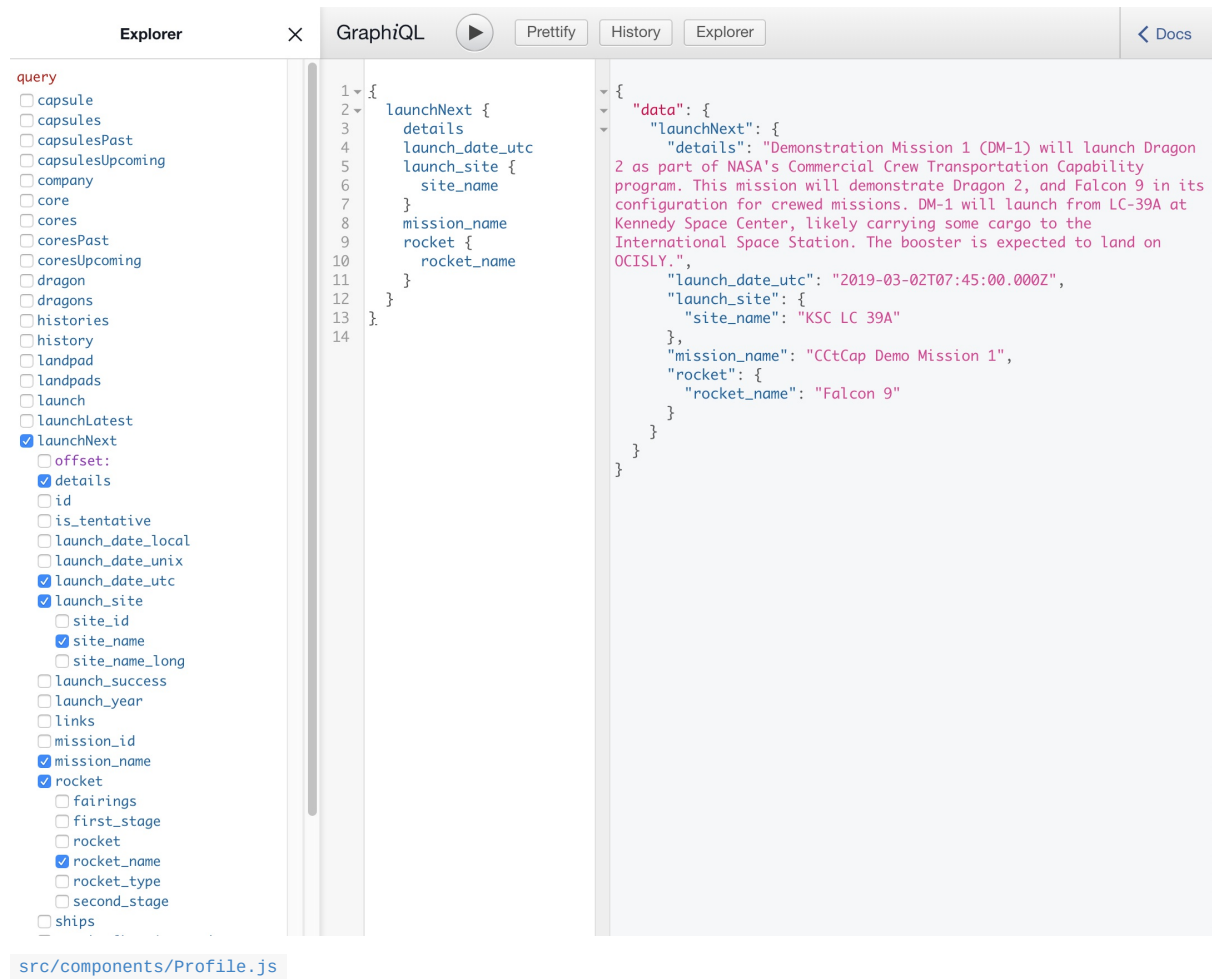
```

<Query
  query={LAUNCH_QUERY}
  client={apolloSpace}
>

```

For building the `LAUNCH_QUERY`, let's see what data is available from the API by browsing its GraphQL:

[api.spacex.land/graphql/](https://api.spacex.land/graphql/). From the available queries, it looks like the relevant one for us is `launchNext`, and we can pick a few fields to display:



```
import gql from 'graphql-tag'

const LAUNCH_QUERY = gql`
  query LaunchQuery {
    launchNext {
      details
      launch_date_utc
      launch_site {
        site_name
      }
      mission_name
      rocket {
        rocket_name
      }
    }
  }
`
```

Now we can use the `<Query>` —let's put the `button` on the bottom of `Profile`. We need to convert it to a class so that we can have state to toggle whether the launch info is displayed. Then we put the data from the response into a

`<dl>`:

`src/components/Profile.js`

```
import React, { Component } from 'react'
import { Query } from 'react-apollo'

import { apolloSpace } from '../lib/apollo'

class Profile extends Component {
  state = {
```

```

    showLaunch: false
  }

  toggleLaunchVisibility = () => {
    this.setState({ showLaunch: !this.state.showLaunch })
  }

  render() {
    const { user, loggingIn } = this.props

    ...

    <main className="Profile">

      ...

      <div className="Profile-footer">
        <button
          className="Profile-toggle-launch"
          onClick={this.toggleLaunchVisibility}
        >
          <span role="img" aria-label="rocket">

            </span>

          </button>

        {this.state.showLaunch && (
          <Query
            query={LAUNCH_QUERY}
            fetchPolicy="cache-and-network"
            client={apolloSpace}
            onCompleted={() =>
              window.scrollTo({ top: 1000, left: 0, behavior: 'smooth' })
            }
          >
            {({
              data: {
                launchNext: {
                  details,
                  launch_date_utc,
                  launch_site,
                  mission_name,
                  rocket
                }
              } = {}
            },
              loading
            ) =>
              loading ? (
                <div className="Spinner" />
              ) : (
                <div>
                  The next SpaceX launch will be:
                  <dl>
                    <dt>Date</dt>
                    <dd>
                      <code>{new Date(launch_date_utc).toString()}</code>
                    </dd>

                    <dt>Mission</dt>
                    <dd>
                      <code>{mission_name}</code>
                    </dd>

                    <dt>Rocket</dt>
                    <dd>
                      <code>{rocket.rocket_name}</code>
                    </dd>

                    <dt>Launch site</dt>
                    <dd>
                      <code>{launch_site.site_name}</code>
                    </dd>

                    <dt>Details</dt>

```

```

        <dd className="-non-code">{details}</dd>
      </dl>
    </div>
  )
}
</Query>
}}
</div>
</main>
}
}

```

When the button is clicked, the launch info appears below, but since (depending on our screen height and browser settings) we might be at the bottom of the page already, we might not be able to see the info unless we scroll. It would be nice UX to autoscroll down to show the info. `<Query>` has an `onCompleted` prop that is called after the query results are provided to us and our component has re-rendered, so we can call `window.scrollTo` then.

In order to deconstruct `launchNext` we need to add `= {}`, as it will be undefined initially.

We're using `fetchPolicy="cache-and-network"` instead of the default `cache-first` to make sure we always have the latest results. If a user checked the next launch, left the browser open for a while, and checked back later, it's possible that the launch we have in the cache will be old—either the launch already happened, or the plans changed. With `cache-and-network`, `<Query>` will first provide us with the cache data, then send the request to the server, then provide us with the response data. However, something unexpected is now happening when we repeatedly toggle the launch info. Do you notice it?

Every time we show the launch info, it shows the loading spinner. Which we wouldn't expect, because after the first time, it should be immediately giving us data from the cache. If we log `data` and `loading` to see what's going on, we'll find that `data` is always filled, and `loading` is first `true` and then `false`. That's right—at first, `data` is filled at the same time that `loading` is `true`. This is the first time that's happened in our app—always before, `data` has been an empty object while `loading` was `true`. This actually isn't a bug—it's how `loading` is meant to work. It's true whenever a request is currently in flight.

If we just want to display data whenever it's available, we can test whether there's data instead of using `loading`:

```

{({
  data: {
    launchNext: {
      details,
      launch_date_utc,
      launch_site,
      mission_name,
      rocket
    } = {}
  }
}) =>
  details ? (
    <div>
      The next SpaceX launch will be:
      ...
    </div>
  ) : (
    <div className="Spinner" />
  )
}

```

Now we'll only see the spinner the first time.

We're done! We can add more SpaceX data to different parts of our app by importing `apolloSpace` and using it in Apollo's `client` prop. And we can add more APIs by creating more `ApolloClient` instances.

## Extended topics

Section contents:

- [Linting](#)
  - [Setting up linting](#)
  - [Fixing linting errors](#)
  - [Using linting](#)
- [Uploading files](#)
- [Testing](#)

## Linting

If you're jumping in here, `git checkout 26_0.2.0` (tag [26\\_0.2.0](#), or compare [26...27](#))

Linters analyze code for errors without running the code—they just look at the code. [ESLint](#) is the main linter for JavaScript. It's already being used in our app by Create React App. However, their ESLint settings just cover JavaScript—they don't check our GraphQL queries to see if they're valid. Let's set that up!

First let's run ESLint as it's currently set up. We have a script in our `package.json` that just runs `eslint src/`:

```
$ npm run lint

> guide@0.2.0 lint /guide
> eslint src/
```

It doesn't print out any linting errors. We can check the exit code to make sure:

```
$ echo $?
0
```

In Mac and Linux, each program has an **exit code**. In Bash, we can print out the last exit code with `echo $?`. An exit code of `0` means success.

## Setting up linting

The npm package `eslint-plugin-graphql` (already in our `package.json` `devDependencies`) adds support for GraphQL to ESLint. We can tell ESLint to use it by modifying our config file:

`.eslintrc.js`

```
module.exports = {
  extends: 'react-app',
  plugins: ['graphql'],
  parser: 'babel-eslint',
  rules: {
    'graphql/template-strings': [
      'error',
      {
        schemaJson: require('./schema.json')
      }
    ]
  },
}
```

- `extends: 'react-app'` : Use Create React App's rules as a base
- `plugins: ['graphql']` : Use `eslint-plugin-graphql`

- `schemaJson: require('./schema.json')` : Look in the current directory for the schema

What schema? We want ESLint to validate our queries against our API's schema—the one the `api.graphql.guide` server has, that Playground shows us in the SCHEMA tab. It makes sense that ESLint is going to need it. But how do we get it in a JSON file? There's a tool on npm called `graphql-cli` that we can use to download it. It's in our `devDependencies`, and the program name is `graphql`. Our `update-schema` script uses it:

```
"update-schema": "graphql get-schema -e https://api.graphql.guide/graphql -o schema.json"
```

- `-e` sets the endpoint
- `-o` gives the output file name

So we can run `npm run update-schema`, and now we have a `schema.json`. It's like a verbose form of what we see in the Playground SCHEMA tab, and starts with:

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutation"
      },
      "subscriptionType": {
        "name": "Subscription"
      },
      "types": [
        {
          "kind": "OBJECT",
          "name": "Query",
          "description": "",
          "fields": [
            {
              "name": "sections",
              "description": "",
              "args": [
                {
                  "name": "lastCreatedAt",
                  "description": "",
                  "type": {
                    "kind": "SCALAR",
                    "name": "Float",
                    "ofType": null
                  },
                  "defaultValue": null
                }
              ],
              "name": "limit",
              "description": "",
              "type": {
                "kind": "SCALAR",
                "name": "Int",
                "ofType": null
              },
              "defaultValue": null
            }
          ],
          "name": "sections",
          "description": "",
          "args": [
            {
              "name": "lastCreatedAt",
              "description": "",
              "type": {
                "kind": "SCALAR",
                "name": "Float",
                "ofType": null
              },
              "defaultValue": null
            }
          ],
          "name": "limit",
          "description": "",
          "type": {
            "kind": "SCALAR",
            "name": "Int",
            "ofType": null
          },
          "defaultValue": null
        }
      ],
      "name": "Query",
      "description": "",
      "args": [
        {
          "name": "lastCreatedAt",
          "description": "",
          "type": {
            "kind": "SCALAR",
            "name": "Float",
            "ofType": null
          },
          "defaultValue": null
        }
      ],
      "name": "limit",
      "description": "",
      "type": {
        "kind": "SCALAR",
        "name": "Int",
        "ofType": null
      },
      "defaultValue": null
    }
  ],
  "name": "Query",
  "description": "",
  "args": [
    {
      "name": "lastCreatedAt",
      "description": "",
      "type": {
        "kind": "SCALAR",
        "name": "Float",
        "ofType": null
      },
      "defaultValue": null
    }
  ],
  "name": "limit",
  "description": "",
  "type": {
    "kind": "SCALAR",
    "name": "Int",
    "ofType": null
  },
  "defaultValue": null
}
]
```

We can see that a `__schema` has `types` that include an object with `name: "Query"` with a field named `sections` which has `args` `lastCreatedAt` and `limit`. And if we scroll down, we see more familiar fields and types.

## Fixing linting errors

Now we can try running ESLint again:

```
$ npm run lint
```

```

> guide@0.2.0 lint /Users/me/gh/guide
> eslint src/

/Users/me/gh/guide/src/components/CurrentTemperature.js
  80:5  error  Cannot query field "weather" on type "Query"  graphql/template-strings

/Users/me/gh/guide/src/components/Profile.js
  155:5  error  Cannot query field "launchNext" on type "Query"  graphql/template-strings

/Users/me/gh/guide/src/components/Section.js
  163:9  error  Cannot query field "scrolly" on type "Section"    graphql/template-strings
  175:7  error  Cannot query field "scrolly" on type "Section"    graphql/template-strings
  188:9  error  Cannot query field "scrolly" on type "Section"    graphql/template-strings
  204:9  error  Cannot query field "scrolly" on type "Section"    graphql/template-strings
  221:5  error  Cannot query field "setSectionScroll" on type "Mutation"  graphql/template-strings

/Users/me/gh/guide/src/lib/apollo.js
  71:3  error  The Query definition is not executable  graphql/template-strings

/Users/me/gh/guide/src/lib/withUser.js
  18:5  error  Cannot query field "loginInProgress" on type "Query"  graphql/template-strings

✖ 9 problems (9 errors, 0 warnings)

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! guide@0.2.0 lint: `eslint src/`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the guide@0.2.0 lint script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
npm ERR!     /Users/me/.npm/_logs/2019-03-04T01_50_08_741Z-debug.log

```

We get a lot of errors! And we can see that the exit code is no longer `0`:

```

npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! guide@0.2.0 lint: `eslint src/`
npm ERR! Exit status 1

```

Exit status `1` means that the exit code of the command `eslint src/` was `1`.

Let's go through the errors. First up:

```

/Users/me/gh/guide/src/components/CurrentTemperature.js
  80:5  error  Cannot query field "weather" on type "Query"  graphql/template-strings

```

which is referring to:

`src/components/CurrentTemperature.js`

```

const TEMPERATURE_QUERY = gql`
  query TemperatureQuery {
    weather(lat: $lat, lon: $lon)
      @rest(
        type: "WeatherReport"
        path: "weather?appid=4fb00091f111862bed77432aead33d04&{args}"
      ) {
        main
      }
  }
`

```

ESLint is looking at our `schema.json` and not finding `weather` as a top-level Query field. Of course it's not! `weather` isn't part of the Guide API—it's from our [weather REST API](#). So we don't want this query linted against the schema. We can tell ESLint to ignore this file by adding `/* eslint-disable graphql/template-strings */` to the top of the file. Now if we re-run `npm run lint`, we no longer see that error.

8 errors left to go! The next is:

```
/Users/me/gh/guide/src/components/Profile.js
155:5 error Cannot query field "launchNext" on type "Query" graphql/template-strings
```

`launchNext` is from our query to the SpaceX API, which of course has a different schema from the rest of our queries. So far we've only told ESLint about `schema.json`, the Guide API schema. But `eslint-plugin-graphql` does support multiple schemas. The way it determines what strings to parse as GraphQL is by the template literal tag name (`gql`). We can use a different tag name for the SpaceX query and have that tag be checked against a different schema. Let's use `spaceql` instead of our current `gql`:

`src/components/Profile.js`

```
import spaceql from 'graphql-tag'

const LAUNCH_QUERY = spaceql`
  query LaunchQuery {
    launchNext {
      details
      launch_date_utc
      launch_site {
        site_name
      }
      mission_name
      rocket {
        rocket_name
      }
    }
  }
`
```

And we update the config file:

`.eslintrc.js`

```
module.exports = {
  extends: 'react-app',
  plugins: ['graphql'],
  parser: 'babel-eslint',
  rules: {
    'graphql/template-strings': [
      'error',
      {
        schemaJson: require('./schema.json')
      },
      {
        tagName: 'spaceql',
        schemaJson: require('./spacex.json')
      }
    ]
  }
}
```

We added this object:

```
{
  tagName: 'spaceql',
  schemaJson: require('./spacex.json')
}
```



Which says, “for any GraphQL document created with the template literal tag name `spaceql`, validate it against the schema located in `spacex.json`.” We can get `spacex.json` with `npm run update-schema-spacex`:

```
"update-schema-spacex": "graphql get-schema -e https://api.spacex.land/graphql -o spacex.json"
```

And now when we lint, we get one fewer error! The next set of errors is:

```
/Users/me/gh/guide/src/components/Section.js
163:9  error  Cannot query field "scrollY" on type "Section"      graphql/template-strings
175:7  error  Cannot query field "scrollY" on type "Section"      graphql/template-strings
188:9  error  Cannot query field "scrollY" on type "Section"      graphql/template-strings
204:9  error  Cannot query field "scrollY" on type "Section"      graphql/template-strings
221:5  error  Cannot query field "setSectionScroll" on type "Mutation"  graphql/template-strings
```

`scrollY` is the piece of [local state](#) in our `Section` queries:

`src/components/Section.js`

```
const NEXT_SECTION_QUERY = gql`
  query NextSection($id: String!) {
    section(id: $id) {
      id
      next {
        id
        content
        views
        scrollY @client
      }
    }
  }
`
```

And `setSectionScroll` is our [local mutation](#). ESLint will find neither of these in the Guide API schema. We can suppress the errors by adding this line to the top of the file:

```
/* eslint-disable graphql/template-strings */
```

2 more errors to go! Here's the next:

```
/Users/me/gh/guide/src/lib/apollo.js
71:3  error  The Query definition is not executable  graphql/template-strings
```

This refers to the `Query` in our local state schema:

```
const typeDefs = gql`
  type Query {
    loginInProgress: Boolean
  }
  type Mutation {
    setSectionScroll(id: String!, scrollY: Int!): Boolean
  }
`
```

Instead of `eslint-disable` ing the whole file, let's just disable part of it. That way if we later add a document to a different part of the file, it will be linted.

`src/lib/apollo.js`

```
...

/* eslint-disable graphql/template-strings */
```

```
const typeDefs = gql`
  type Query {
    loginInProgress: Boolean
  }
  type Mutation {
    setSectionScroll(id: String!, scrollY: Int!): Boolean
  }
`
/* eslint-enable graphql/template-strings */

...
```

And the last error is another local state field—let's just put the `eslint-disable` comment at the top of `src/lib/withUser.js`.

```
/Users/me/gh/guide/src/lib/withUser.js
18:5  error  Cannot query field "loginInProgress" on type "Query"  graphql/template-strings
```

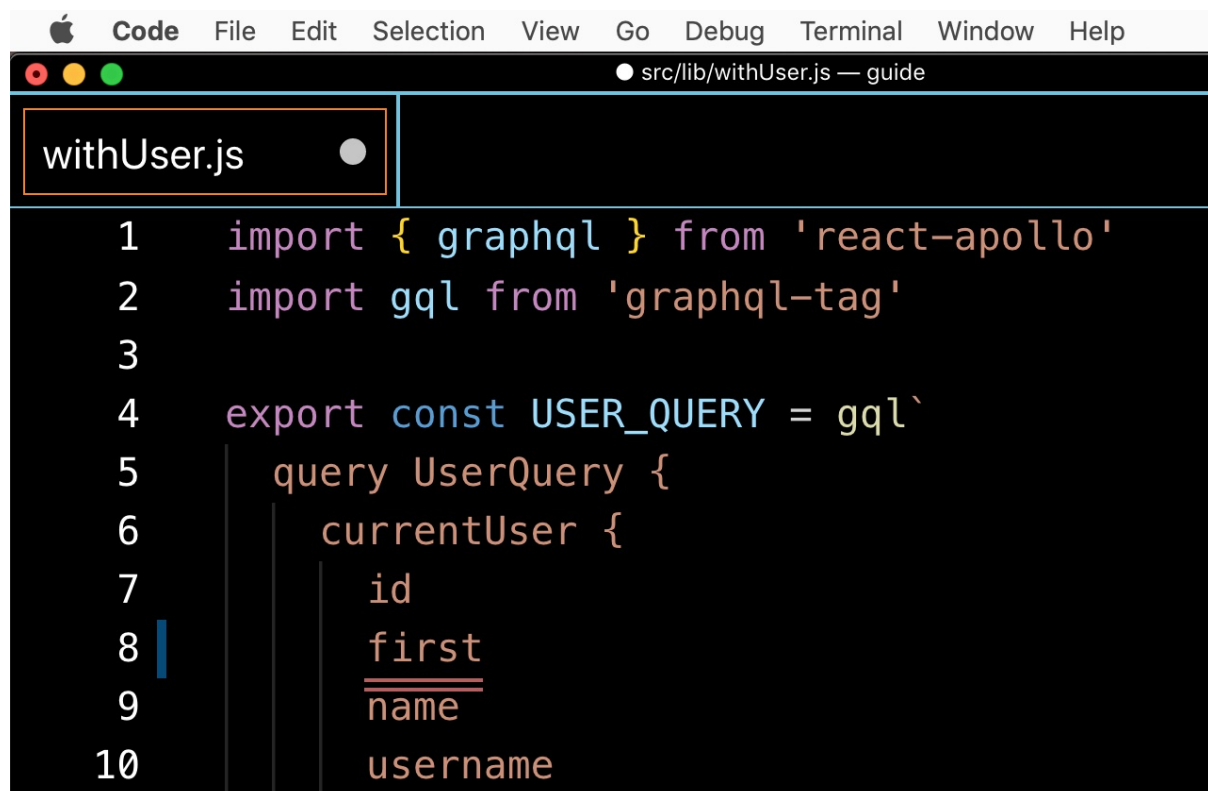
## Using linting

Usually people don't manually run `npm run lint` on the command line. Instead, they set up one or more of the following, which all automatically run the linter:

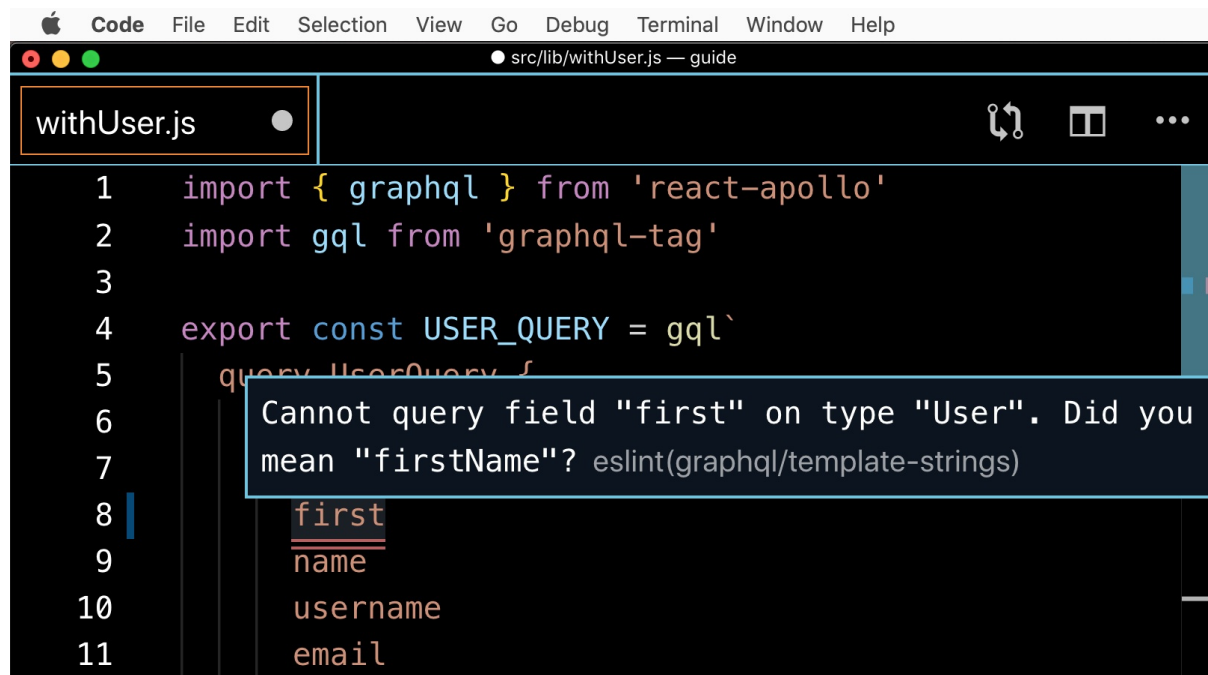
- [Editor integration](#)
- [Pre-commit hook](#)
- [Continuous integration](#)

## Editor integration

Most editors have a linting plugin. VSCode has this [ESLint plugin](#). It looks for a configuration file in the current workspace (for us it would find `.eslintrc.js`) and runs ESLint in the background whenever we type something new into the editor. For instance if we type in `first` as a field of `currentUser`, it is underlined:



And if we hover over the word, we see the linting error:



Cannot query field "first" on type "User". Did you mean "firstName"?

Since ESLint has the schema, it knows that `currentUser` resolves to a `User`, and that `first` isn't one of the fields of the `User` type. When we change it to `firstName`, the error underline goes away.

Some linting errors have automatic fixes, and we can have the plugin make those fixes whenever we save the file by enabling this setting:

```
"eslint.autoFixOnSave": true
```

## Pre-commit hook

Git has [a lot of hooks](#)—times when git will run a program for you. One such hook is pre-commit. A pre-commit hook will be called when a dev enters `git commit` and before git actually does the committing. If the hook program ends with a non-zero exit code, the commit will be canceled. The best way to set up git hooks in our project is with [Husky](#). To do that, we would:

```
npm install husky --save-dev
```

And add to our `package.json`:

```

{
  "husky": {
    "hooks": {
      "pre-commit": "npm run lint"
    }
  }
}

```

Then if we tried to commit but `npm run lint` failed, the commit would be canceled, and we would see the ESLint output with the problem(s) we need to fix.

## Continuous integration

Background: [Continuous integration \(CI\)](#)

Our CI server can do `npm run lint` as one of its tests, prevent deployment if linting fails, display a build failure symbol next to the commit or PR, and link to its site where we can view the error output.

## Uploading files

Background: [CDN](#)

There are two ways to do file uploads: client-side and server-side. In client-side uploads, the client sends the file directly to a cloud service that stores the files. In server-side, the client sends the file to our server, which then stores it someplace (either on a hard drive or with a cloud service—usually the latter). For ease of coding, we recommend client-side. The only possible downside is that someone could upload a lot of files to our service, costing us more money. However, in the unlikely event that this becomes a problem, there are ways with most services to make sure only logged-in users can upload.

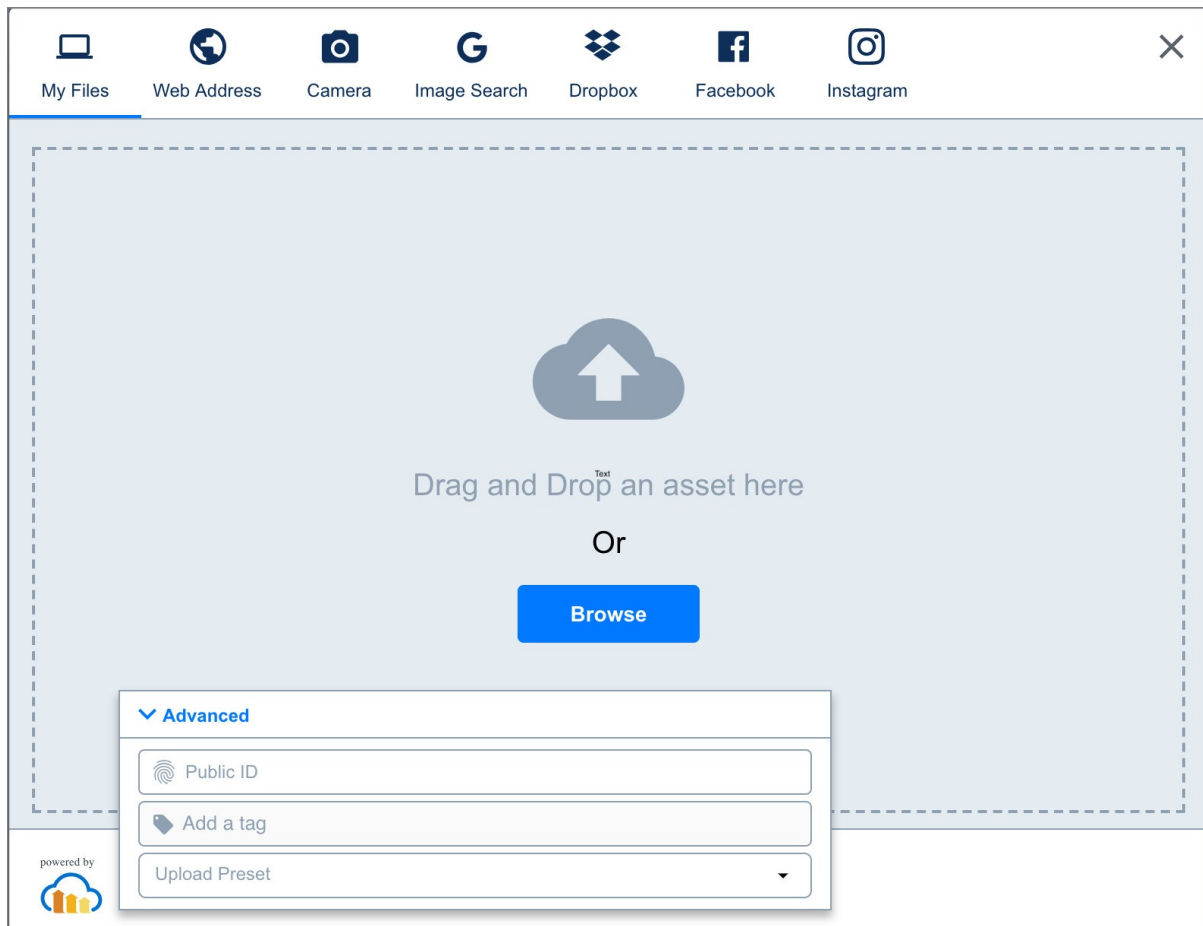
The two main services we recommend are:

- Cloudinary (file storage, CDN, and media file processor)
- Amazon S3 (file storage) and CloudFront ([CDN](#))

Usually an app needs to process images or videos—resizing an image, centering on a face and cropping it, brightening, etc—before using them. For these apps, we recommend Cloudinary as the all-in-one solution. If you're just saving files that need to be stored, and maybe downloaded later unchanged, then S3 is fine.

### Client-side

There are two ways to upload to Cloudinary from the client—we can use their upload UI, or we can create our own. Here's what [theirs](#) looks like:



When we open the widget, we give it a callback. The user uses the widget to upload a file to our Cloudinary account, and the widget calls our callback, providing us the ID of the file as an argument. We send the ID to our server in a mutation, and our server saves it to our database. We use the ID to construct the URL of the file, for example:

```
http://res.cloudinary.com/graphql/guide/file-id.jpg
```

If we want our own UI, we can render a file input styled however we want, and then we POST the input file to the Cloudinary server [like in this React example](#). (And then, as before, we get an ID back to send in a mutation to the server.)

## Server-side

Here's what we would do to upload a file to our server:

```
npm install apollo-upload-client
```

apollo.js

```
import { ApolloClient } from 'apollo-client'
import { InMemoryCache } from 'apollo-cache-inmemory'
import { createUploadLink } from 'apollo-upload-client'

const client = new ApolloClient({
  cache: new InMemoryCache(),
  link: createUploadLink({
    uri: 'https://api.graphql.guide/graphql'
  })
})
```

## FileUpload.js

```
import gql from 'graphql-tag'
import { Mutation } from 'react-apollo'

const UPLOAD_FILE_MUTATION = gql`
  mutation UploadFile($file: Upload!) {
    uploadFile(file: $file) {
      id
      fileName
    }
  }
`

const FileUpload = () => (
  <Mutation mutation={UPLOAD_FILE_MUTATION}>
    {mutate => (
      <input
        type="file"
        required
        onChange={({
          target: {
            validity,
            files: [file]
          }
        }) => validity.valid && mutate({
          variables : { file }
        })}
      />
    )}
  </Mutation>
)

export default FileUpload
```

Our server needs to support the [GraphQL multipart request spec](#). We'll see in the server chapter how to do this in Node using [graphql-upload](#).

## Testing

We're holding off on writing this section until the hooks+suspense version of React Apollo comes out. For now, we recommend the built-in `<MockedProvider>` for the easiest setup or [this approach](#) for the most succinct test code.

We also recommend using [Jest](#) and [react-testing-library](#). If you'd like a video introduction to them, as well as testing in general, we recommend [this course](#).

## Chapter 7: Vue

We haven't gotten to this chapter yet! For now, check out [this tutorial](#).

## Chapter 8: React Native

We haven't gotten to this chapter yet, but most things from the [React chapter](#) apply to React Native. For now, here's a great [small example](#), and a [paid course](#).



## Chapter 9: iOS

We haven't gotten to this chapter yet! For now, check out the [Apollo iOS docs](#)

## Chapter 10: Android

We haven't gotten to this chapter yet! For now, check out the [Apollo Android docs](#).

# Chapter 11: Server Dev

Chapter contents:

- [Introduction](#)
  - [Why build a GraphQL server?](#)
  - [What kind of GraphQL server should I build?](#)
- [Building](#)
  - [Project setup](#)
  - [Types and resolvers](#)
  - [Authenticating](#)
  - [Data sources](#)
    - [Setting up](#)
    - [File structure](#)
    - [Creating reviews](#)
  - [Custom scalars](#)
  - [Creating users](#)
    - [Protecting with secret key](#)
    - [Setting user context](#)
    - [Linking users to reviews](#)
  - [Authorizing](#)
  - [Errors](#)
    - [Nullability](#)
    - [Union errors](#)
    - [formatError](#)
      - [Logging errors](#)
      - [Masking errors](#)
    - [Error checking](#)
    - [Custom errors](#)
  - [Subscriptions](#)
    - [githubStars](#)
    - [reviewCreated](#)
- [Testing](#)
  - [Static testing](#)
  - [Review integration tests](#)
  - [Code coverage](#)
  - [User integration tests](#)
  - [Unit tests](#)
  - [End-to-end tests](#)
- [Production](#)
  - [Deployment](#)
    - [Options](#)
    - [Deploying](#)
    - [Environment variables](#)
  - [Database hosting](#)
    - [MongoDB hosting](#)
    - [Redis hosting](#)
      - [Redis PubSub](#)
      - [Redis caching](#)

- Querying in production
- Analytics
- Error reporting
- More data sources
  - SQL
    - SQL setup
    - SQL data source
    - SQL testing
  - REST
  - GraphQL
  - Custom data source
  - Prisma
- Extended topics
  - Mocking
  - Pagination
    - Offset-based
    - Cursors
      - after an ID
      - Relay cursor connections
  - File uploads
    - Client-side
    - Server-side
  - Schema validation
  - Apollo federation
    - Federated service
    - Federated gateway
    - Extending entities
    - Managed federation
    - Deploying federation
  - Hasura
  - Schema design
    - One schema
    - User-centric
    - Easy to understand
    - Easy to use
    - Mutations
      - Arguments
      - Payloads
    - Versioning
  - Custom schema directives
    - @tshirt
    - @upper
    - @auth
  - Subscriptions in depth
    - Server architecture
    - Subscription design
  - Security
    - Auth options
      - Authentication
      - Authorization
    - Denial of service

- [Performance](#)
    - [Data fetching](#)
    - [Caching](#)
  - [Future](#)
- 

## Introduction

Background: [HTTP](#), [Server](#)

Welcome to the server chapter! This is the last—and longest—chapter. We'll learn most of the concepts through building the Guide API server, which backs the apps we built in the client chapters. The server will primarily store data in MongoDB, but we'll also connect to several other data sources, including SQL and REST. We'll write it in JavaScript, but all server-side GraphQL libraries use [the same execution method](#), and most of the concepts in this chapter will apply to writing GraphQL servers in other languages. To see the differences, check out these backend tutorials:

- [Java](#)
- [Python](#)
- [Ruby](#)
- [Scala](#)
- [Elixir](#)

There are also GraphQL libraries in these languages:

- [.NET](#)
- [Clojure](#)
- [Go](#)
- [PHP](#)

This chapter is split into five parts:

- **Introduction**
- [Building](#)
- [Production](#)
- [More data sources](#)
- [Extended topics](#)

In **Building**, we build a GraphQL server from scratch, including authentication and authorization, query and mutation resolvers that talk to a database, error handling, subscriptions, and testing. In **Production**, we deploy our server and update it with things that are helpful to have in production, like error reporting, analytics, and security against attack. In **More data sources**, we connect our server to other databases and a REST API. In **Extended topics**, we learn about various new server-side topics and go into more depth on previous topics like the schema, subscriptions, and auth.

## Why build a GraphQL server?

There are three main reasons why we might decide our server should be a GraphQL server:

1. So we can use GraphQL on the client and gain all the client-side benefits of GraphQL.
2. To simplify our server code: instead of setting up many endpoints and implementing fetching and formatting logic for each, we set up one endpoint and write a single resolver for each data type.
3. To avoid having to create new endpoints or new APIs in the future.

For coders, #1 and #2 are often the most compelling, because it improves our quality of life 😊. For companies, #3 is often the most compelling, since they save time and money: they get a single, flexible API that covers all their business data, which means that instead of having to create new endpoints or entire APIs for new features or apps, they can just use their existing GraphQL API (and in some cases add fields and resolvers).

## What kind of GraphQL server should I build?

Actually, the first choice we have is whether to build it or generate it 😊. There are services that can save us a lot of time by generating a production-ready GraphQL backend for us. We'll go over the pros/cons and how to set one up in the [Hasura section](#).

If we do decide to build our own server, there are two situations we might be in:

1. **Existing project**, in which case we'll either be adding a GraphQL layer in front of our existing servers, or adding a GraphQL endpoint to existing servers.
2. **New project** (a.k.a. *greenfield*), in which case we have a choice of which architecture to use.

There are two main architectures:

1. **Microservices** (a collection of servers that each cover a different business capability). GraphQL as the API gateway: the client talks to the GraphQL server API gateway, which talks to services (via GraphQL, REST, gRPC, Thrift, etc), which talk to databases.
2. **Monolith** (a single server that covers all business logic). GraphQL as the application layer: the client talks to the GraphQL server, which talks directly to databases.

Microservices are in vogue and the word “monolith” is often used with a scornful tone, but in most cases, it's better to have a monolith. Martin Fowler, one of the leaders in software design, [wrote](#):

So my primary guideline would be don't even consider microservices unless you have a system that's too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.

While there are a lot of huge tech companies that use microservices and are better off for it, they're better off because they're huge—not because microservices are a general good practice.

If we have an existing monolith, it often makes sense to add a GraphQL endpoint to that server instead of putting a GraphQL server in front of the monolith. For example, if we have an Express monolith that has a lot of thin REST routes that call model functions that contain the business logic and data fetching, then it would be easy to add a `/graphql` route with `apollo-server-express` and implement resolvers that call the same model functions as the REST routes. Or if all of our logic was in the routes themselves, and we didn't need to continue supporting the REST API, we could move the code we needed over to resolvers and [Apollo data sources](#).

When we're adding a GraphQL layer in front of an existing backend, whether it's a microservices or monolith backend, we can make the choice between continuing to develop the existing backend or gradually moving logic to the GraphQL layer. If we're doing microservices and want to keep that architecture, then it's easy to keep implementing services (in whatever language(s) we implement services) and either extend the GraphQL schema and resolvers or use [schema federation](#).

Another question is what language to write our GraphQL server in. In the case of adding to an existing monolith, we'll use the GraphQL server library for the same language. In all other cases (new projects or a GraphQL layer in front of existing microservices or monoliths), we generally recommend JavaScript. It's by far the most popular type of GraphQL server, and has thus developed the best ecosystem of libraries and services.

The server we'll be creating in this chapter is a greenfield monolith, so it will talk directly to the database. However, most of the concepts will carry over to the microservice model. The largest difference will be either:

- using schema federation to combine multiple GraphQL services
- fetching data and resolving mutations by talking to the services (e.g. with REST) instead of the database

We'll go over both of these options later in the chapter.

## Building

Background: [Node & npm & nvm](#), [git](#), [JavaScript](#)

- [Project setup](#)
- [Types and resolvers](#)
- [Authenticating](#)
- [Data sources](#)
  - [Setting up](#)
  - [File structure](#)
  - [Creating reviews](#)
- [Custom scalars](#)
- [Creating users](#)
  - [Protecting with secret key](#)
  - [Setting user context](#)
  - [Linking users to reviews](#)
- [Authorizing](#)
- [Errors](#)
  - [Nullability](#)
  - [Union errors](#)
  - [formatError](#)
    - [Logging errors](#)
    - [Masking errors](#)
  - [Error checking](#)
  - [Custom errors](#)
- [Subscriptions](#)
  - [githubStars](#)
  - [reviewCreated](#)

We're using Node because it's the most popular platform for GraphQL Servers and has the best ecosystem.

JavaScript is also the most used programming language in the world! ↻:arrow\_up:

## Project setup

There are a few different things to set up when starting a new Node project. We've set them up in branch `0` of our server repo, [github.com/GraphQLGuide/guide-api](https://github.com/GraphQLGuide/guide-api):

```
$ git clone https://github.com/GraphQLGuide/guide-api.git
$ cd guide-api/
$ git checkout 0_0.2.0
$ npm install
```

We now have these files:

```
.babelrc
.git/
.gitignore
.nvmrc
.prettierrc
node_modules/
package-lock.json
package.json
```

Let's look at each to see what they're for.

- `.babelrc` :

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "node": "12.0.0"
        }
      }
    ]
  ],
  "plugins": ["import-graphq1"]
}
```

`@babel/preset-env` transpiles JavaScript to work in the target environment—where we'll be running the code. In chapter 6, that was the browser. For this chapter, the target environment is Node. We'll target version `12.0.0` so that the transpiled code will work in that or higher versions.

- `.git/` : directory where git stores its data.
- `.gitignore` :

```
node_modules/
dist/
```

A list of which files and folders we don't want committed to git. We don't want `node_modules/` as they're added when we `npm install`. And `dist/` will be generated by the build script in our `package.json`.

- `.nvmrc` :

```
12
```

The file that tells nvm which version of Node to use. `12` means the latest stable `12.*` version.

- `.prettierrc` :

```
singleQuote: true
semi: false
trailingComma: none
arrowParens: avoid
```

Because single quotes and no semicolons is the One True Way to style JavaScript.

Just kidding—there isn't one right way to style code. This is just author Loren's preference 😊.

And the last two settings were the Prettier default when this chapter was written.

- `node_modules/` : directory to which npm downloads all of the packages our code depends on.
- `package-lock.json` : precise current versions of all the packages.



- `package.json` :

```
{
  "name": "guide-api",
  "version": "0.1.0",
  "description": "api.graphql.guide",
  "scripts": {
    "dev": "nodemon -e js,graphql --exec 'npm run update-graphql-imports && babel-node src/index.js'",
    "start": "node dist/index.js",
    "build": "babel src -d dist --ignore **/*.test.js",
    "update-graphql-imports": "rm -rf ./node_modules/.cache/@babel"
  },
  "engines": {
    "node": ">=12"
  },
  "dependencies": {
    "@sentry/node": "5.15.5",
    "apollo-datasource-mongodb": "0.2.6",
    "apollo-datasource-rest": "0.8.1",
    "apollo-server": "2.12.0",
    "apollo-server-cache-redis": "1.1.6",
    "apollo-server-testing": "2.12.0",
    "aws-sdk": "2.666.0",
    "casual": "1.6.2",
    "datasource-sql": "1.3.0",
    "date-fns": "2.12.0",
    "dotenv": "8.2.0",
    "graphql": "14.6.0",
    "graphql-redis-subscriptions": "2.2.1",
    "graphql-request": "1.8.2",
    "graphql-tools": "4.0.8",
    "ioredis": "4.16.3",
    "join-monster": "2.1.1",
    "join-monster-graphql-tools-adapter": "0.1.0",
    "jsonwebtoken": "8.5.1",
    "jwks-rsa": "1.8.0",
    "knex": "0.21.1",
    "lodash": "4.17.15",
    "mongodb": "3.5.7",
    "sqlite3": "4.2.0"
  },
  "devDependencies": {
    "@babel/cli": "7.8.4",
    "@babel/core": "7.9.6",
    "@babel/node": "7.8.7",
    "@babel/preset-env": "7.9.6",
    "apollo-link": "1.2.14",
    "apollo-link-http": "1.5.17",
    "babel-plugin-import-graphql": "2.7.0",
    "eslint": "6.8.0",
    "eslint-plugin-node": "11.1.0",
    "husky": "4.2.5",
    "jest": "25.5.2",
    "node-fetch": "2.6.0",
    "nodemon": "2.0.3"
  },
  "homepage": "https://github.com/GraphQLGuide/guide-api",
  "repository": {
    "type": "git",
    "url": "git+https://github.com/GraphQLGuide/guide-api"
  },
  "bugs": {
    "url": "https://github.com/GraphQLGuide/guide-api/issues"
  },
  "private": true,
  "author": "The GraphQL Guide <hi@graphql.guide> (https://graphql.guide)"
}
```

Let's look at the scripts first:

```
"scripts": {
```

```

"dev": "nodemon -e js,graphql --exec 'npm run update-graphql-imports && babel-node src/index.js'",
"start": "node dist/index.js",
"build": "babel src -d dist --ignore **/*.test.js",
"update-graphql-imports": "rm -rf ./node_modules/.cache/@babel"
},

```

- `npm run dev` will watch our JS and GraphQL files, and whenever one of them changes, it will transpile them with Babel and run them with Node.
- `npm start` will start our server in production using the transpiled version of our code located in `dist/`.
- `npm run build` will transpile our code from `src/` to `dist/` (ignoring test files).
- `npm update-graphql-imports` is used by `npm run dev` to clear the babel GraphQL plugin cache.

The `engines` attribute, similar to `preset-env`'s `target`, describes where the code is meant to be run. For us, it's meant to be run in any version 8 or higher of Node.

```

"engines": {
  "node": ">=12"
},

```

All together, what we've got configured is:

- Git
- npm
- nvm
- Babel
- Prettier

## Apollo Server

If you're jumping in here, `git checkout 0_0.2.0` (tag `0_0.2.0`, or compare `0...1`)

It takes less than 20 lines of JavaScript to get a working GraphQL server up and running! 😊 The best GraphQL server library is `apollo-server`, and here's the basic setup:

`src/index.js`

```

import { ApolloServer, gql } from 'apollo-server'

const server = new ApolloServer({
  typeDefs: gql`
    type Query {
      hello: String!
    }
  `,
  resolvers: {
    Query: {
      hello: () => '        '
    }
  }
})

server
  .listen({ port: 4000 })
  .then(({ url }) => console.log(`GraphQL server running at ${url}`))

```

The main export of `apollo-server` is `ApolloServer`, and its two required parameters are:

- `typeDefs`: our schema, created with the `gql` template literal tag.
- `resolvers`: an object of `resolver` functions that match our schema in structure. Each type— `Query`, `Mutation`, `User`, `Chapter`, etc.—is a top-level attribute, and the next level is that type's field names.

We start the server by calling `.listen()` on a port. When it's done starting up, the Promise is resolved with the URL—in our case, `http://localhost:4000`.

We can test it out with our `dev` script:

```
$ npm run dev

> guide-api@0.1.0 dev /guide-api
> babel-watch src/index.js

GraphQL server running at http://localhost:4000/
```

After a moment, the program gets to the last line of our code, which logs `Server running at http://localhost:4000/`. When we edit `src/index.js`—for instance by changing `console.log`—the server gets restarted:

```
$ npm run dev

> guide-api@0.1.0 dev /guide-api
> babel-watch src/index.js

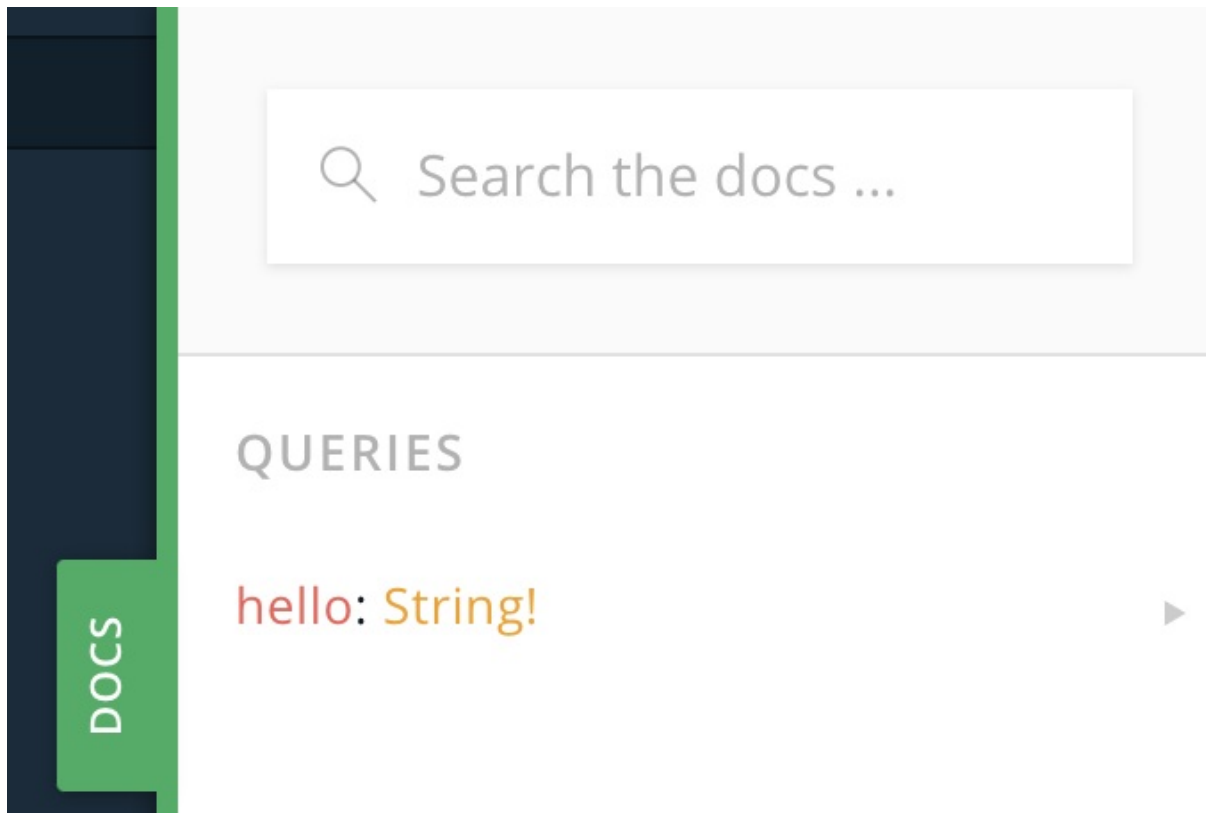
GraphQL server running at http://localhost:4000/
>>> RESTARTING <<<
GraphQL server running with new console.log statement at http://localhost:4000/
```

To stop the server, we can press the `control-c` key combination.

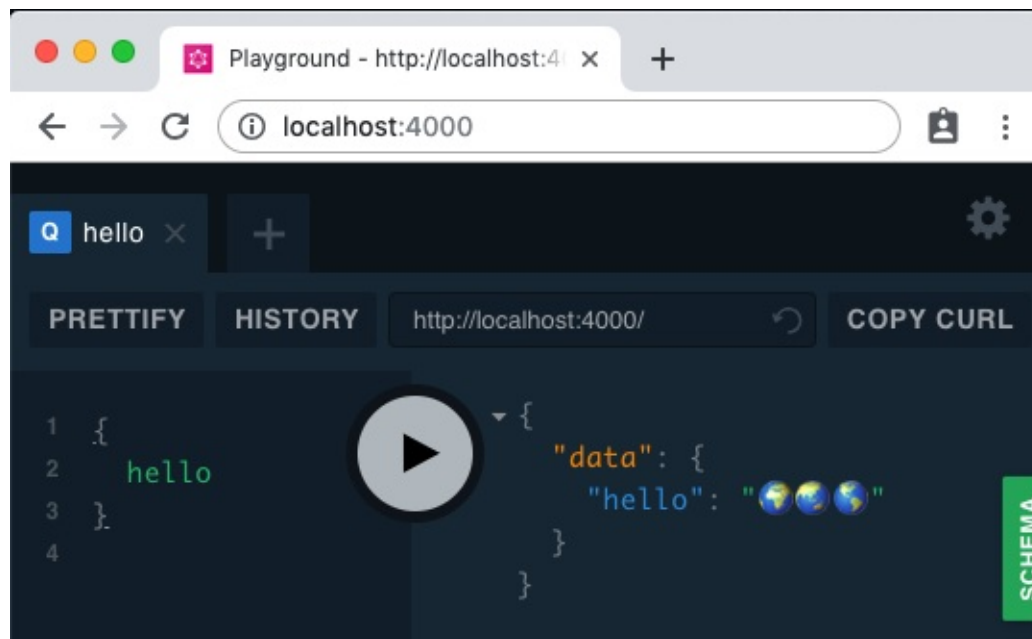
So we've got the server running, but does it work? Let's open up the URL in a browser:

<http://localhost:4000>

By default Apollo Server loads GraphQL Playground, an IDE for writing GraphQL queries. We can see our tiny schema by clicking on “DOCS” on the right:



And we can test out our one query:



And it works!

## Types and resolvers

If you're jumping in here, `git checkout 1_0.2.0` (tag `1_0.2.0`, or compare `1...2`)

The heart of a GraphQL server is the types and resolvers. The schema has the types and each type's fields, and the resolvers *resolve* each field. We generally resolve fields by fetching data from a data source, formatting fetched data, or enacting mutations.

Let's add some more types and fields to get a better sense of how they match up with resolvers. We want people to be able to submit reviews for the book, so we need a mutation:

```
type Mutation {
  createReview(text: String!, stars: Int): Review
}
```

The convention for naming a creation mutation is `create<Type>`, and it usually resolves to that type (hence `: Review` at the end). However, it's best practice to use a single input type as an argument instead of listing out all the scalars needed. So let's change it to:

```
type Mutation {
  createReview(input: CreateReviewInput!): Review
}
input CreateReviewInput {
  text: String!
  stars: Int
}
```

We also want people to be able to read past reviews, so we add a Query field:

```
type Query {
  hello: String!
  reviews: [Review!]!
}
```

We don't have a `Review` type yet, so we need to add that:

```
type Review {
  text: String!
  stars: Int
  fullReview: String!
}
```

All together, our new schema looks like this:

`src/index.js`

```
const server = new ApolloServer({
  typeDefs: gql`
    type Query {
      hello: String!
      reviews: [Review!]!
    }
    type Review {
      text: String!
      stars: Int
      fullReview: String!
    }
    type Mutation {
      createReview(review: CreateReviewInput!): Review
    }
    input CreateReviewInput {
      text: String!
      stars: Int
    }
  `,
  resolvers: { ... }
})
```

We need a resolver for each field except for the `input` type. (Input types are only used for mutation arguments: fields can't resolve to input types, so input types don't need resolvers.) The structure of our `resolvers` object matches the schema, so it should look like:

```
const server = new ApolloServer({
  typeDefs: ...
  resolvers: {
    Query: {
      hello: () =>
      reviews: () =>
    },
    Review: {
      text: () =>
      stars: () =>
      fullReview: () =>
    }
    Mutation: {
      createReview: () =>
    },
  },
})
```

Now let's fill them in! We'll start with `createReview` :

```
const reviews = [
  {
    text: 'Super-duper book.',
    stars: 5
  }
]

const server = new ApolloServer({
  typeDefs: ...
```

```

    resolvers: {
      ...
      Mutation: {
        createReview: (_, { review }) => {
          reviews.push(review)
          return review
        }
      }
    }
  })
}

```

We don't need the first resolver parameter, just the second, which contains the mutation argument—the review. We add it to our array of reviews and return it (since our schema says that `createReview` resolves to an object of type `Review`).

Next we can implement the `reviews` Query field:

```

const reviews = [
  {
    text: 'Super-duper book.',
    stars: 5
  }
]

const server = new ApolloServer({
  typeDefs: ...
  resolvers: {
    Query: {
      hello: ...
      reviews: () => reviews
    }
  }
})

```

For `query.reviews` we just return our array of reviews. But a GraphQL server doesn't just return the `reviews` array to the client: it looks at the schema, sees that `Query.reviews` resolves to `[Review!]!`, checks to make sure the `reviews` array is non-null, and then resolves each object in the array as a `Review`. The way it does that is by calling `Review` field resolvers, which we also have to define:

```

const reviews = [
  {
    text: 'Super-duper book.',
    stars: 5
  }
]

const server = new ApolloServer({
  typeDefs: ...
  resolvers: {
    Query: {
      hello: ...
      reviews: () => reviews
    },
    Review: {
      text: review => review.text
      stars: review => review.stars
      fullReview: review =>
        `Someone on the internet gave ${review.stars} stars, saying: "${review.text}"`
    }
  }
})

```

When the GraphQL server calls a `Review` field resolver, it provides the object as the first parameter, for example:

```
{
  text: 'Super-duper book.',
  stars: 5
}
```

The `text` and `stars` type fields we can just resolve to the corresponding object properties (for example, `text: review => review.text`). And we can actually take the `text` and `stars` resolvers out, because Apollo Server will do that by default. The `fullReview` field isn't a property on the object, so the default resolver won't work. So we define our own resolver, returning a string constructed from the review's properties.

All together, without the extraneous object property resolvers, we have:

`src/index.js`

```
import { ApolloServer, gql } from 'apollo-server'

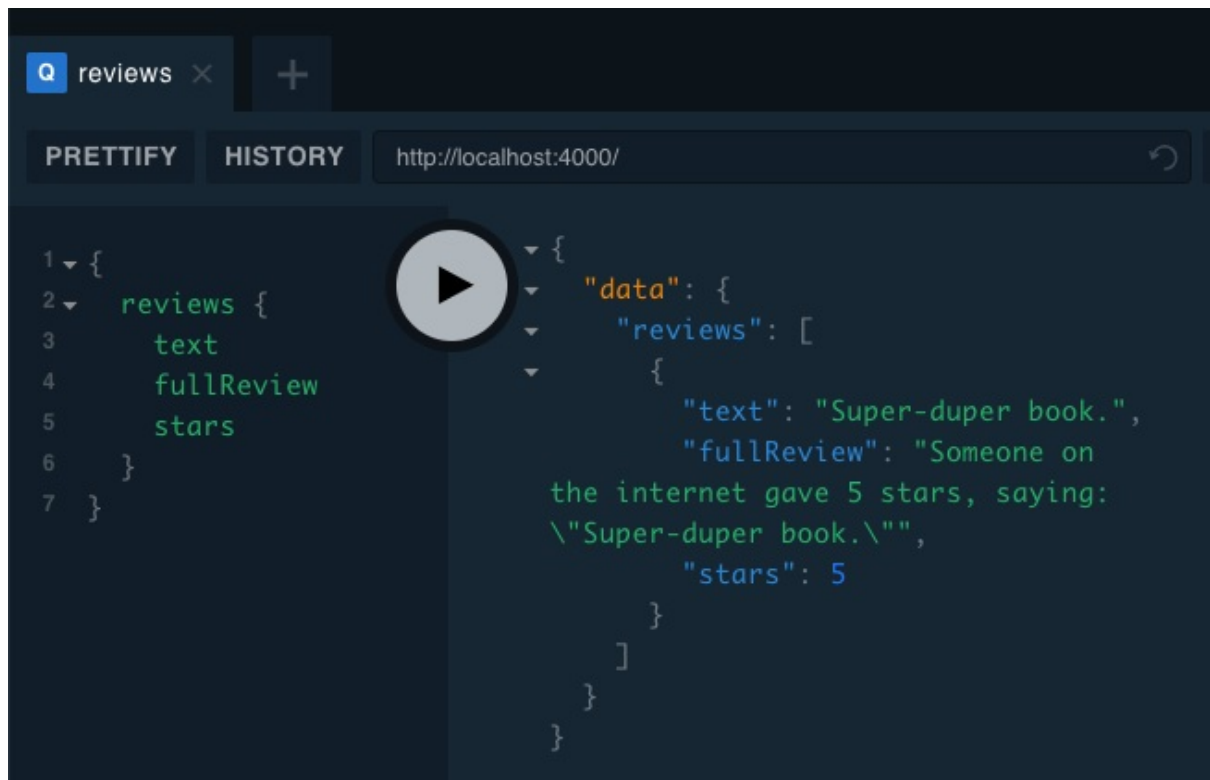
const reviews = [
  {
    text: 'Super-duper book.',
    stars: 5
  }
]

const server = new ApolloServer({
  typeDefs: gql`
    type Query {
      hello: String!
      reviews: [Review!]!
    }
    type Review {
      text: String!
      stars: Int
      fullReview: String!
    }
    type Mutation {
      createReview(review: CreateReviewInput!): Review
    }
    input CreateReviewInput {
      text: String!
      stars: Int
    }
  `,
  resolvers: {
    Query: {
      hello: () => 'hello',
      reviews: () => reviews
    },
    Review: {
      fullReview: review =>
        `Someone on the internet gave ${review.stars} stars, saying: "${review.text}"`
    },
    Mutation: {
      createReview: (_, { review }) => {
        reviews.push(review)
        return review
      }
    }
  },
})

server
  .listen({ port: 4000 })
  .then(({ url }) => console.log(`GraphQL server running at ${url}`))
```

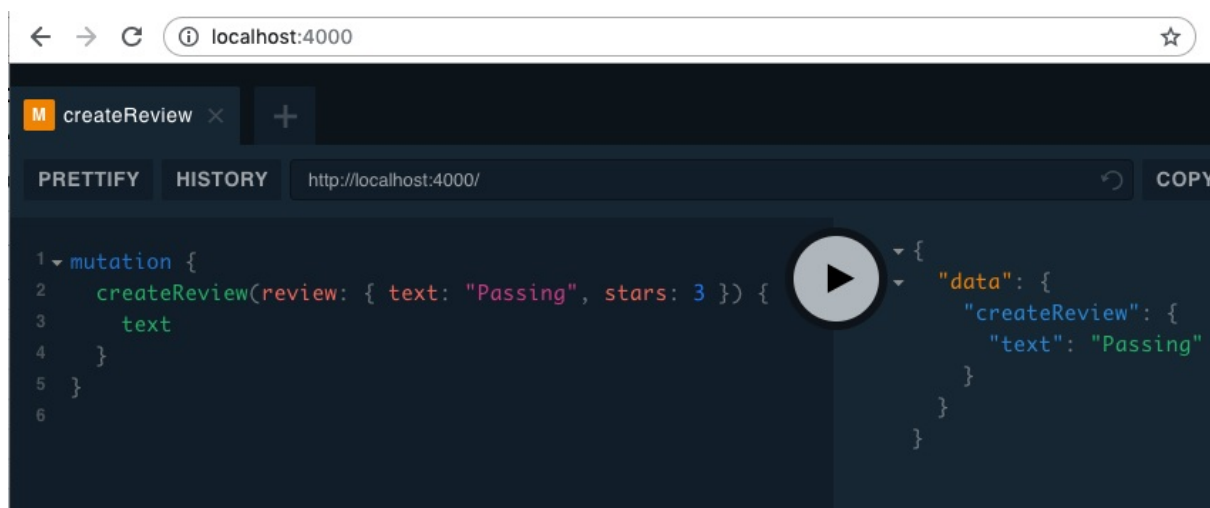
We can try it out with `npm run dev`, see that Playground loads, and try out the new query:

```
{
  reviews {
    text
    fullReview
    stars
  }
}
```



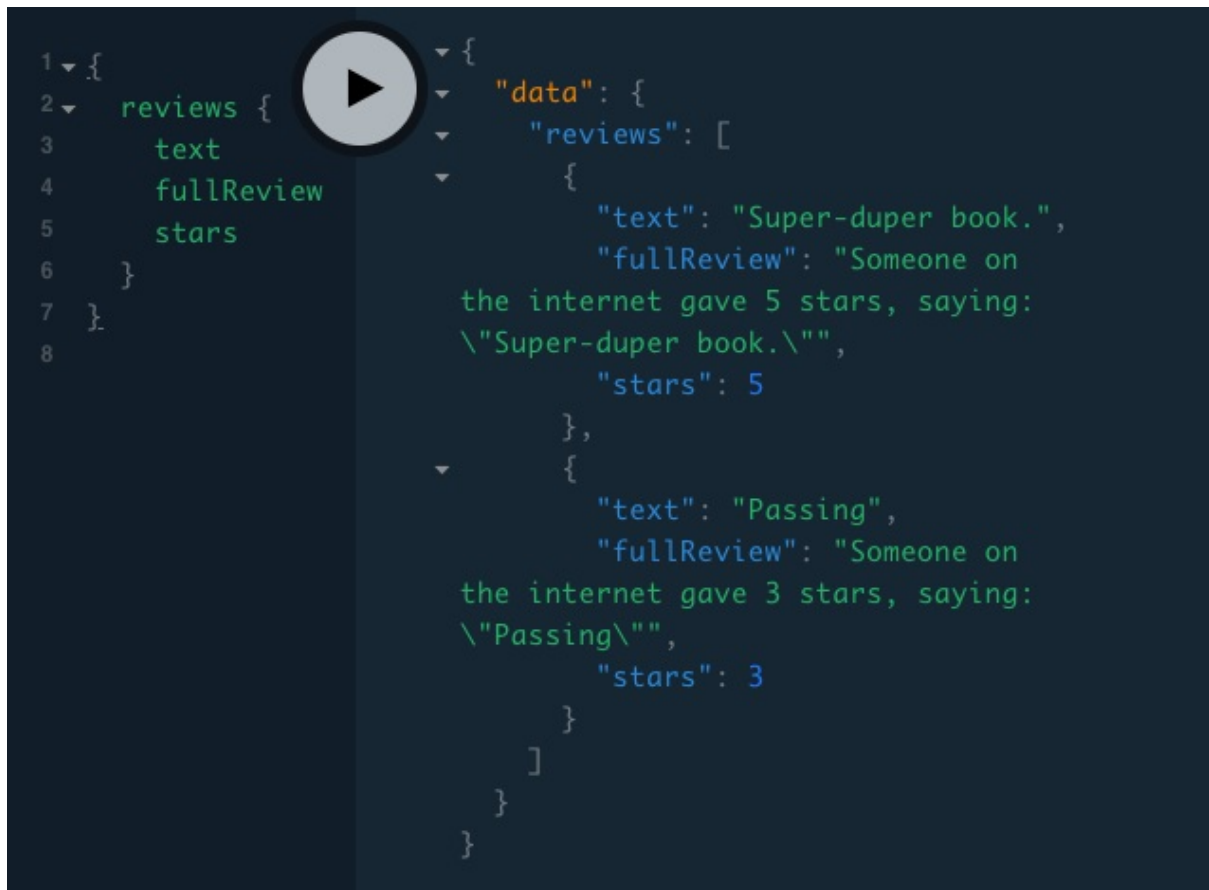
localhost:4000: { reviews { text fullReview stars } }

We see our one hard-coded review. Now if we do our mutation followed by the `reviews` query, we'll see both that and the new review:



localhost:4000: mutation { createReview(review: { text: \"Passing\", stars: 3 }) { text } }





```
localhost:4000: { reviews { text fullReview stars } }
```

Notice how the only things we changed in our server were our types (in the Apollo Server `typeDefs` parameter) and our resolvers. These two things (including code called by our resolver functions) will be the bulk of the coding we do for our GraphQL server.

## Authenticating

Background: [Authentication](#)

If you're jumping in here, `git checkout 2_0.2.0` (tag [2\\_0.2.0](#), or compare [2...3](#))

One thing that's done outside of types and resolvers is creating *context*, which is an object provided to resolvers. We set context using the `context` of `ApolloServer()`. The `context` param is either an object or, more commonly, a function that returns an object. The function is called at the beginning of every request. The most common use of the `context` function is authenticating the user making the request and adding their info to the context. Here's an example with a hard-coded user:

```
src/index.js
```

```
const server = new ApolloServer({
  typeDefs: gql`
    type Query {
      me: User
      ...
    }
    type User {
      firstName: String
      lastName: String
    }
  `
  ...
})
```

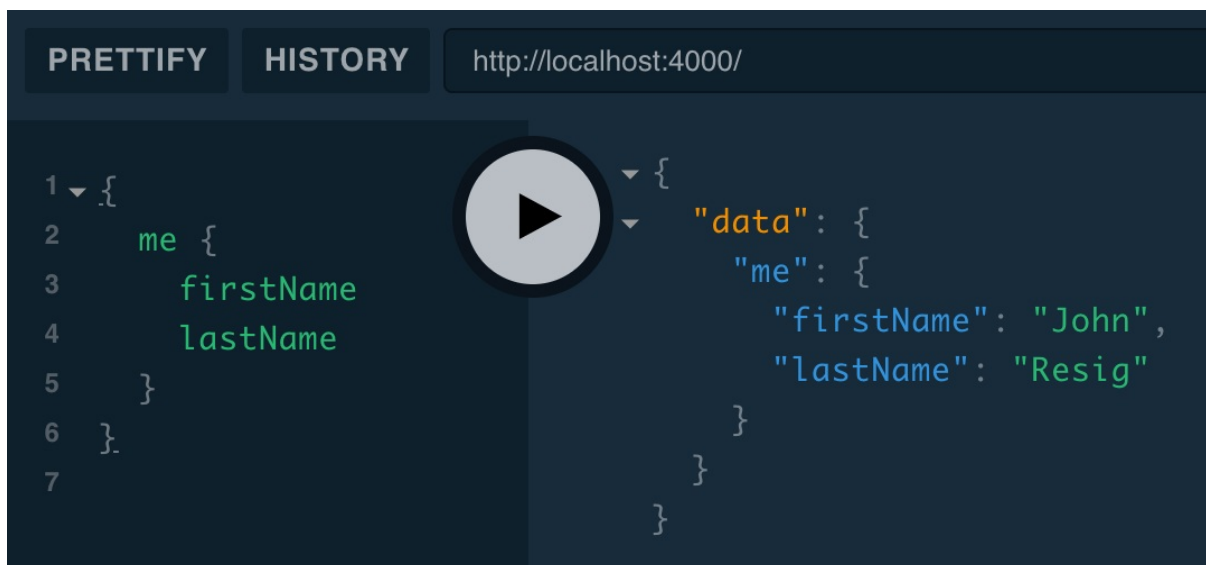
```

    },
    resolvers: {
      Query: {
        me: (_, __, context) => context.user,
        ...
      },
      ...
    },
    context: () => {
      const user = {
        firstName: 'John',
        lastName: 'Resig'
      }

      return { user }
    }
  })
})

```

Context is resolvers' third parameter. For the `me` resolver, we just return the `user` property. We can try it out:



localhost:4000: { me { firstName lastName } }

Now let's figure out the real user. The Guide uses JWTs stored in LocalStorage, so authentication is done by cryptographically verifying the token provided in the request's authorization header. We get the request as an argument to the `context function`:

```

import { getAuthIdFromJWT } from '../util/auth'

const server = new ApolloServer({
  ...
  context: async ({ req }) => {
    const context = {}

    const jwt = req.headers.authorization
    const authId = await getAuthIdFromJWT(jwt)
    console.log(authId)

    return context
  }
})

```

`getAuthIdFromJWT()` verifies the given JWT and returns what we're calling the user's *authId*—a unique string identifying the user that we get as the OpenID subject ( `verifiedToken.sub` below). Here's the function's implementation:

`src/util/auth.js`

```

import jwt from 'jsonwebtoken'
import jwks from 'jwks-rsa'
import { promisify } from 'util'

const verify = promisify(jwt.verify)

const jwksClient = jwks({
  cache: true,
  rateLimit: true,
  jwksUri: 'https://graphql.auth0.com/.well-known/jwks.json'
})

const getPublicKey = (header, callback) => {
  jwksClient.getSigningKey(header.kid, (e, key) => {
    callback(e, key.publicKey || key.rsaPublicKey)
  })
}

export const getAuthIdFromJWT = async token => {
  if (!token) {
    return
  }

  const verifiedToken = await verify(token, getPublicKey, {
    algorithms: ['RS256'],
    audience: 'https://api.graphql.guide',
    issuer: 'https://graphql.auth0.com/'
  })

  return verifiedToken.sub
}

```

It calls `verify()` from the `jsonwebtoken` package. In order to verify, it needs the Guide's public signing key. To get that, we use the `jwks-rsa` package.

Now if we send a `{ hello }` query in Playground, we see `undefined` in the server logs. `authId` is undefined because `req.headers.authorization` is undefined. Which means that Playground isn't sending an authorization header with our query. We can set it by clicking "HTTP HEADERS" in the bottom-left to open the JSON headers section. We want to set the authorization header to our JWT, but how do we get that? It's produced by Auth0 during the login process and saved to local storage, so we can get it by logging in at [graphql.guide/me](https://graphql.guide/me), opening the console, and entering:

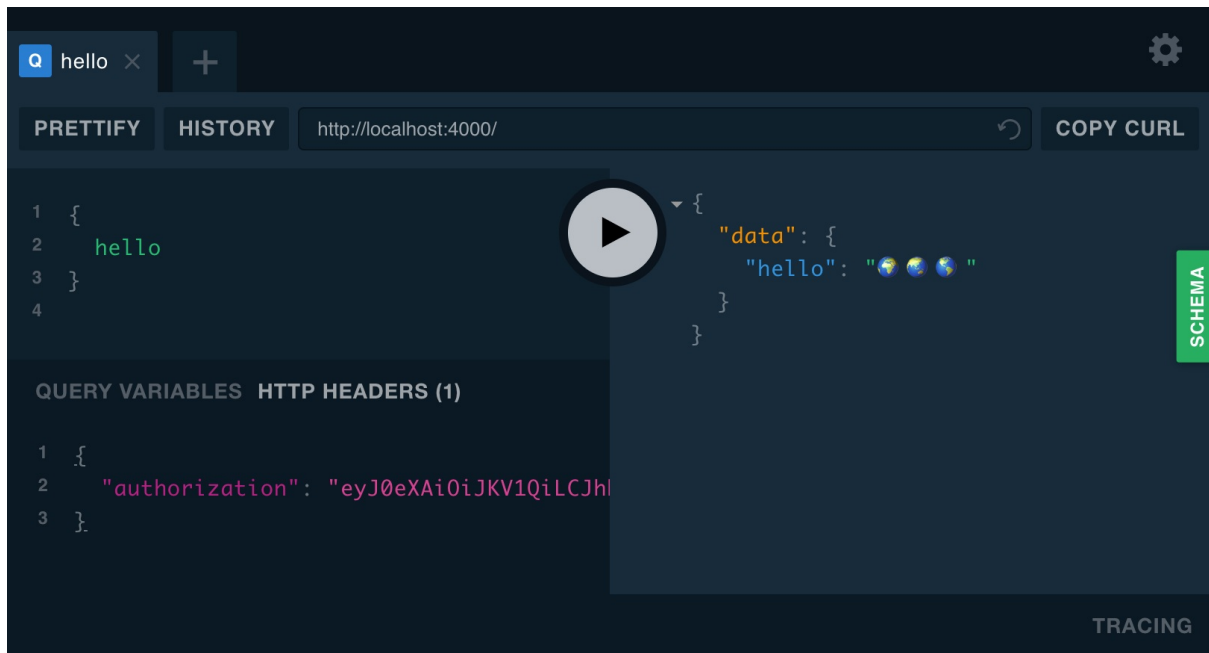
```
localStorage.getItem('auth.accessToken')
```

And it prints our JWT! It's a long, random-looking, mostly alphanumeric string with some periods, dashes, and underscores. We can copy it to the Playground headers section:

```

{
  "authorization": "your JWT here"
}

```



If you get a `jwt malformed` error, you likely didn't copy the whole token. Try opening the Application tab in Chrome dev tools, selecting `auth.accessToken`, and copying from the value panel at the bottom of the window.

Make note of your authorization header—you'll need it for making queries in other sections of this chapter.

Now when we run the query, we see our `authId` logged—something like this:

```
$ npm run dev
> guide-api@0.1.0 dev /guide-api
> babel-watch src/index.js

GraphQL server running at http://localhost:4000/
undefined
github|1615
```

The format is `github|N`, where `N` is our primary key in the users table of GitHub's database. (It's an incrementing integer, which means that author John was GitHub's 1,615th user! 😊)

The next thing that should happen in the code is looking up the user in our database—something like:

```
context: async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  context.user = await db.collection('users').findOne({ authId })

  return context
}
```

But we don't have a database set up yet (we'll set it up in the next section and add users in [Setting user context](#)), so let's just test whether the `authId` is ours (replacing the strings with your own):

```
context: async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  if (authId === 'github|1615') {
    context.user = {
```

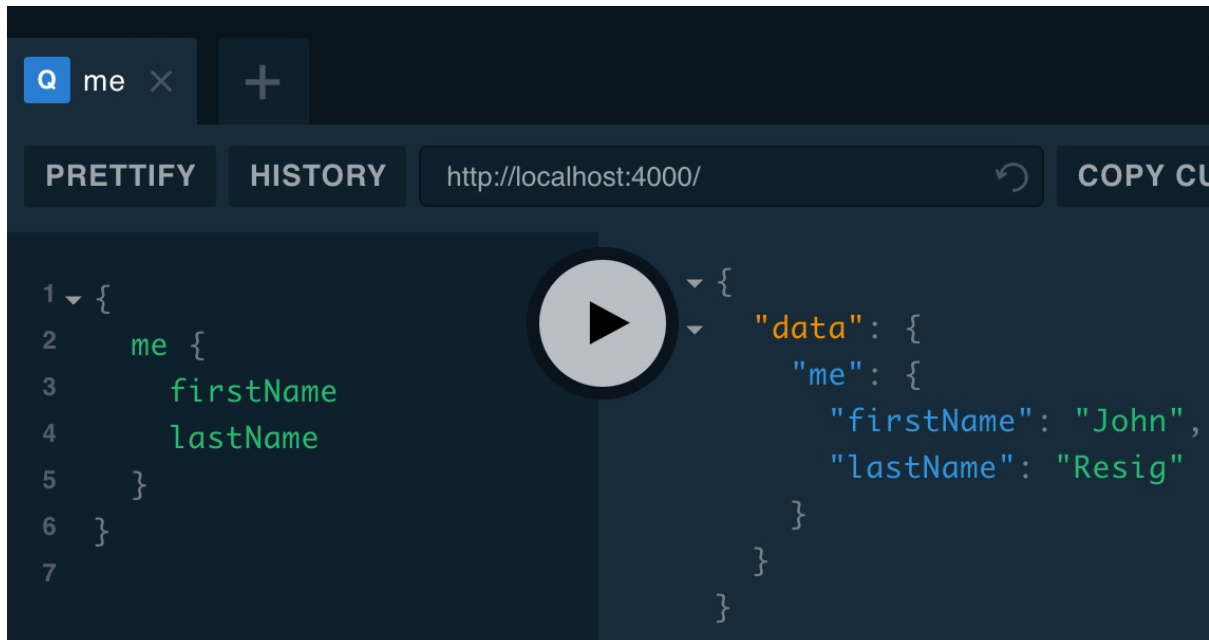
```

    firstName: 'John',
    lastName: 'Resig'
  }
}

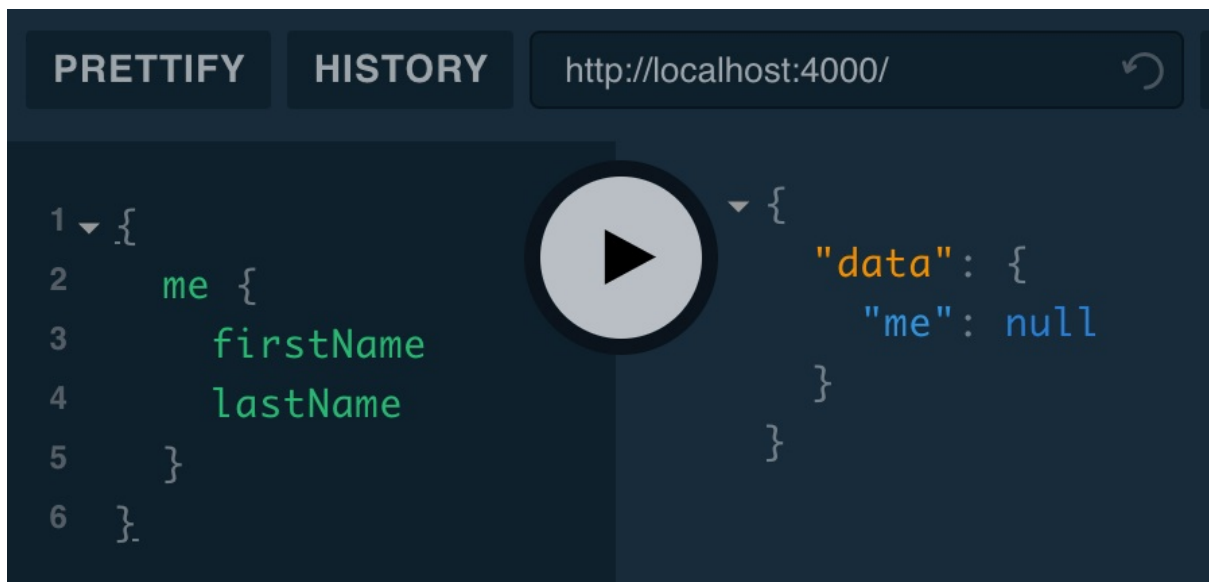
return context
}

```

Now if we do a `me` query with our authorization header, we get our name:



But if we remove the header, we get null:



This is because the `query.me` resolver returns `context.user`, which is not defined.

In this section we learned how to put our JWT in the authorization header, verify it on the server, add the user to context, and access the context in resolvers. In the next section we'll look at connecting to a database and creating users.

## Data sources

- [Setting up](#)
- [File structure](#)
- [Creating reviews](#)

## Setting up

Background: [MongoDB](#), [JavaScript classes](#)

If you're jumping in here, `git checkout 3_0.2.0` (tag [3\\_0.2.0](#), or compare [3...4](#))

Our reviews are currently stored in a JavaScript array variable. There are a few problems with this storage method. JavaScript variables are part of the Node server process, which means that:

- When the server process restarts (for instance when we deploy), our reviews get erased.
- When the server machine loses power (it's unlikely but possible for our data center to have a power outage), the data kept in RAM (which requires electricity to remember things) is lost. Since each process's variables are stored in RAM, our reviews get erased.
- When we have multiple server processes (common in the age of Heroku, when it's easy to scale up small containers), the user will see different reviews based on which container each request is routed to.
- When we we're using serverless and don't have a long-running server process (widely introduced by AWS Lambda in 2014 and now, with Now 2.0 and Netlify Functions, becoming the standard way to host "servers" 😊), the process is started up for each request, so every `reviews` query would return just the single item we started out with.

The solution to all of these problems is to have a database that all of the server processes can talk to—one that stores data on a drive that doesn't require power to remember things (either a disk drive that stores data on magnetic disks or a solid-state drive that stores data in flash memory).

We'll be using MongoDB because it's the most popular database among Node developers and because it's simple to use. The object-based API is easy to understand, and we don't need to create a schema or do migrations. (Of course, just as a schema is useful in GraphQL, it's useful for databases, and we could enforce a schema for our MongoDB database, for example with the [Mongoose ORM](#), but we'll be using the simplest model layer possible.) For an introduction to MongoDB, check out the [MongoDB section](#) of the Background chapter.

There are two main ways to talk to a database from our GraphQL resolvers: [data sources](#) and [Prisma](#). We generally recommend Prisma (a next-generation ORM) for ease of use, and we'll learn how to use it in a [later section](#). For now, we'll use a MongoDB data source, for the same reasons we used Create React App instead of Next.js or Gatsby in the React chapter—data sources are more basic and familiar.

Data sources are classes that interact with a source of data (a database or a service). They often take care of some amount of batching queries and caching responses. We'll go into them more deeply in the [More data sources](#) section.

Usually there are two classes: a superclass that we import from a library that matches our type of database, and a subclass that we implement. There are superclass libraries for MongoDB, [SQL](#), and [REST](#), and we'll also learn how to [create our own](#). The MongoDB library is `apollo-datasource-mongodb`, and its superclass is called `MongoDataSource`. Let's use it to create a data source for a `'reviews'` MongoDB collection:

`src/data-sources/Reviews.js`

```
import { MongoDataSource } from 'apollo-datasource-mongodb'

export default class Reviews extends MongoDataSource {
  all() {
    return this.collection.find().toArray()
  }
}
```

We start with a single method `all()` that fetches all reviews from the collection. Where does `this.collection` come from, you might ask? It's set in the constructor (defined in `MongoDataSource`), which gets the collection as an argument:

```
const reviews = new Reviews(db.collection('reviews'))
```

But in order to do that, we need to set up the database! We can install and start MongoDB on Windows with [these steps](#) or with [Homebrew](#) on a Mac:

```
$ brew tap mongodb/brew
$ brew install mongodb-community
$ brew services start mongodb-community
```

The database is now running on our computer. We connect to it with the `mongodb` package:

`src/db.js`

```
import { MongoClient } from 'mongodb'

export let db

const URL = 'mongodb://localhost:27017/guide'

const client = new MongoClient(URL, { useNewUrlParser: true })
client.connect(e => {
  if (e) {
    console.error(`Failed to connect to MongoDB at ${URL}`, e)
    return
  }

  db = client.db()
})
```

`'mongodb://localhost:27017/'` is the default URL of the MongoDB server running on our computer, and `'guide'` is the name of our database. Now we can import `db` and use it to create our data source. Data sources are created in a function that we pass to `ApolloServer`:

`src/index.js`

```
import Reviews from './data-sources/Reviews'
import { db } from './db'

const server = new ApolloServer({
  typeDefs: ...
  resolvers: ...
  dataSources: () => ({
    reviews: new Reviews(db.collection('reviews'))
  }),
  context: ...
})
```

Like the `context` function, the `dataSources` function is run for each request, so each request gets a new instance of each data source. `ApolloServer` adds data sources to the context so that we can access them in our resolvers like this:

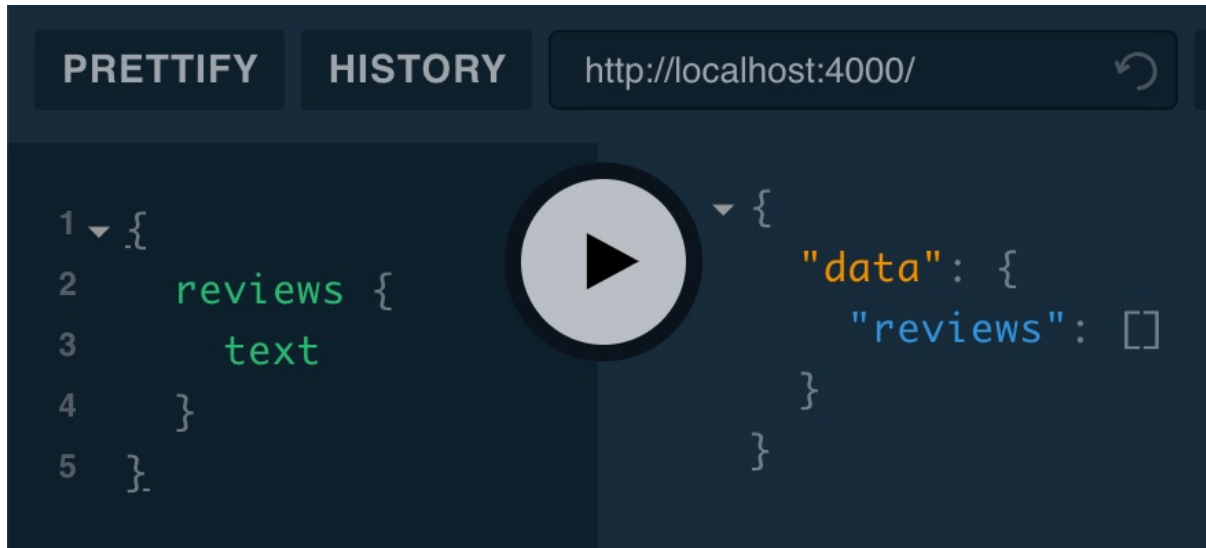
```
const server = new ApolloServer({
  typeDefs: ...
  resolvers: {
    Query: {
      me: (_, __, context) => context.user,
      hello: () => ' ',
      reviews: (_, __, { dataSources }) => dataSources.reviews.all()
```

```

    },
    ...
  },
  dataSources: () => ({
    reviews: new Reviews(db.collection('reviews'))
  }),
  context: ...
})

```

We always get context as the third argument to our resolvers, and here in the `query.reviews` resolver we're destructuring context's `dataSources` property. Then we get the instance of our `Reviews` data source, `dataSources.reviews`, and call its `.all()` method. Now when we do our reviews query again, we get an empty array, since nothing is yet in the `reviews` collection:



## File structure

If you're jumping in here, `git checkout 4_0.2.0` (tag [4\\_0.2.0](#), or compare [4...5](#))

Our `src/index.js` file is getting long, and continuing to put most of our code in one file would get ridiculous 😊. Let's really simplify this file and get our `ApolloServer` creation down to just:

```

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

```

with each parameter imported from other files. There's no one right way to structure the other files, but our favorite is:

- directories for the schema, resolvers, and data sources
- one file for each major type, for example:
  - `schema/Review.graphql` for the `Review` type schema
  - `resolvers/Review.js` for the resolvers associated with the `Review` type
  - `data-sources/Reviews.js` for the `reviews` collection data source

With this structure, our `src/` looks like:

```

.
├── context.js
├── data-sources
│   └── Reviews.js

```



```

├── index.js
├── db.js
├── index.js
├── resolvers
│   ├── Review.js
│   ├── User.js
│   └── index.js
├── schema
│   ├── Review.graphql
│   ├── User.graphql
│   └── schema.graphql
├── util
└── auth.js

```

Some notes on the above:

- We haven't yet made a data source for the users collection.
- We have context in a single file `context.js`, but if that ever got too long, we could make a `context/` directory and split it into multiple files.
- We have `index.js` files so that we can import the directory (for example `import resolvers from './resolvers'` imports from `./resolvers/index.js`).
- We don't have an `index.js` in `schema/` because they're `.graphql` files, and you can't import a directory with GraphQL imports.

For GraphQL imports, we're using a babel plugin called `babel-plugin-import-graphql` which replaces our imported `.graphql` files with schema objects (the same ones that the `gql` template string tag creates). We could have instead done JS files with template strings and given an array of them as our `typeDefs` parameter, which would look like this:

```

// schema/Review.js
import gql from 'graphql-tag'

export default gql`
type Review {
  text: String!
  stars: Int
  fullReview: String!
}
`

// schema/User.js
import gql from 'graphql-tag'

export default gql`
type User {
  firstName: String
  lastName: String
}
`

// schema/index.js
import reviewSchema from './Review.js'
import userSchema from './User.js'

export default [reviewSchema, userSchema]

// index.js
import typeDefs from './schema'

const server = new ApolloServer({
  typeDefs,
  ...
})

```

Instead, we have:

```
src/schema/schema.graphql
```

```

type Query {
  hello: String!
}

# import Review first
#import 'Review.graphql'
#import 'User.graphql'

```

And the babel plugin makes the `#import` statements work, bringing in these files:

`src/schema/Review.graphql`

```

type Review {
  text: String!
  stars: Int
  fullReview: String!
}

extend type Query {
  reviews: [Review!]!
}

type Mutation {
  createReview(review: CreateReviewInput!): Review
}

input CreateReviewInput {
  text: String!
  stars: Int
}

```

`src/schema/User.graphql`

```

type User {
  firstName: String
  lastName: String
}

extend type Query {
  me: User
}

```

`extend type Query` adds fields to the existing `Query` type (which we defined first in `schema.graphql`).

`Review.graphql` is the first to define `Mutation`, so it doesn't use `extend`. And we import it first so that future files we import below can all do `extend type Mutation`. (And we include the `# import Review first` comment in the file so that others—or our future selves 😊—won't change the order.)

Thanks to our babel plugin, our `schema.graphql` can be imported like this:

```
import typeDefs from './schema/schema.graphql'
```

In our `resolvers/` directory we have `Review.js` and `User.js`, which just have the resolvers related to the `Review` and `User` types, respectively:

`src/resolvers/Review.js`

```

export default {
  Query: {
    reviews: (_, __, { dataSources }) => dataSources.reviews.all(),
  },
  Review: {
    fullReview: review =>
      `Someone on the internet gave ${review.stars} stars, saying: "${
        review.text
      }"`
  }
}

```

```

    },
    Mutation: {
      createReview: (_, { review }) => {
        reviews.push(review)
        return review
      }
    }
  }
}

```

src/resolvers/User.js

```

export default {
  Query: {
    me: (_, __, context) => context.user
  }
}

```

We combine them in `index.js` :

src/resolvers/index.js

```

const resolvers = {
  Query: {
    hello: () => ' '
  }
}

import Review from './Review'
import User from './User'

export default [resolvers, Review, User]

```

We can now import all resolvers with:

```
import resolvers from './resolvers'
```

Next up is data sources! We already have `src/data-sources/Review.js` , so all we need is an `index.js` that will combine future data sources with our `Review.js` and export the function that creates new instances:

src/data-sources/index.js

```

import Reviews from './Reviews'
import { db } from './db'

export default () => ({
  reviews: new Reviews(db.collection('reviews'))
})

```

The last thing we want to move out of `src/index.js` is our context function. It's small enough that we can put it in a single file:

src/context.js

```

import { getAuthIdFromJWT } from './util/auth'

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  if (authId === 'github|1615') {
    context.user = {
      firstName: 'John',
      lastName: 'Resig'
    }
  }
}

```

```

    }

    return context
  }

```

This brings our entire `src/index.js` to just:

```

import { ApolloServer } from 'apollo-server'
import typeDefs from '../schema/schema.graphql'
import resolvers from '../resolvers'
import dataSources from '../data-sources'
import context from './context'

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

server
  .listen({ port: 4000 })
  .then(({ url }) => console.log(`GraphQL server running at ${url}`))

```

So clean! ✨

## Creating reviews

If you're jumping in here, `git checkout 5_0.2.0` (tag [5\\_0.2.0](#), or compare [5...6](#))

In [Setting up](#), we updated our `reviews` query to fetch from MongoDB, but our reviews database collection is empty! So let's get reviews into the database. API clients usually find it helpful if we give them an ID for objects we send them, so let's add one to the schema:

`src/schema/Review.graphql`

```

type Review {
  id: ID!
  text: String!
  stars: Int
  fullReview: String!
}

```

Let's update our `createReview` mutation to talk to the database:

`src/resolvers/Review.js`

```

export default {
  ...
  Mutation: {
    createReview: (_, { review }, { dataSources }) =>
      dataSources.reviews.create(review)
  }
}

```

It just calls a method on our data source, which we need to define:

`src/data-sources/Reviews.js`

```

export default class Reviews extends MongoDataSource {
  all() {
    return this.collection.find().toArray()
  }

  create(review) {

```

```

    this.collection.insertOne(review)
    return review
  }
}

```

`createReview` resolves to a `Review`, so we need to return `review`. And it needs to have an ID. MongoDB's `insertOne()` synchronously adds a generated `_id` to the argument we give it, so when we `return review`, `review._id` is filled in. We return before the MongoDB node library talks to the database in order to send a response to the client as quickly as possible. If we wanted to wait until after we knew that the database operation had completed successfully, we could `await` :

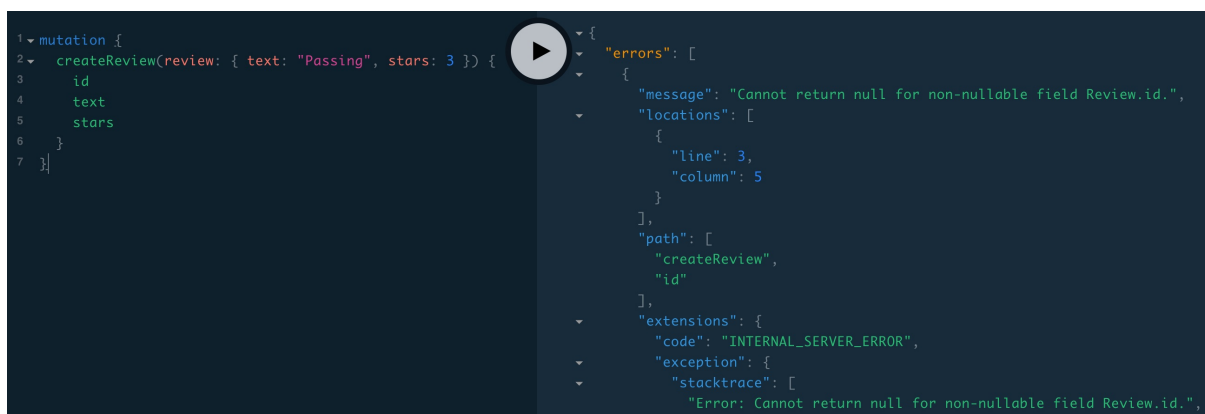
```
async create(review) {
  await this.collection.insertOne(review)
  return review
}
```

In this case, if there were a problem with the database insertion, `insertOne()` would throw an error, which Apollo Server would format and send to the client. Our method is now `async`, which means it returns a Promise, which means our `createReview` resolver returns a Promise. Apollo Server waits for Promises to resolve before continuing the GraphQL execution process.

While it's good that in either case, the `_id` property is added to our `review` object, `_id` doesn't match with our schema (the schema says the `Review` type has a field named `id`, without an underscore). If we create a review and include `id` in the selection set:

```
mutation {
  createReview(review: { text: "Passing", stars: 3 }) {
    id
    text
    stars
  }
}
```

then we get this error:



Apollo Server is trying to resolve the `id` field in our selection set, looking at the review object we return from the `createReview` resolver, and not finding an `id` property on that object. When it can't find a property or `Review` field resolver, it normally returns `null`. However, the `Review` type in our schema has an `!` in the type of `id` (`id: ID!`), so it is non-nullable. Hence the error text: "Cannot return null for non-nullable field Review.id."

We can fix this by adding a `Review.id` resolver:

src/resolvers/Review.js

```
export default {
```

```

Review: {
  id: review => review._id,
  fullReview: review =>
    `Someone on the internet gave ${review.stars} stars, saying: "${
      review.text
    }"`
},
...
}

```

`review._id` is an object—an instance of `ObjectId`, MongoDB's default ID type. `Review.id` is supposed to resolve to the GraphQL `ID` scalar type, which is serialized as a string. This might make us think that we should be getting an error. But if we try our Playground mutation again, it's successful. The reason is that because the schema says the `id` resolver should return an `ID`, Apollo Server knows to call `.toString()` on the object we return.

```

1 mutation {
2   createReview(review: { text: "Passing", stars: 3 }) {
3     id
4     text
5     stars
6   }
7 }

```

```

{
  "data": {
    "createReview": {
      "id": "5cdfb28e48435b90119bd2c6",
      "text": "Passing",
      "stars": 3
    }
  }
}

```

We can now see the list of reviews in the database—one for each time we ran the `createReview` mutation:

```

{
  reviews {
    id
    text
    stars
    fullReview
  }
}

```

```

1 {
2   reviews {
3     id
4     text
5     stars
6     fullReview
7   }
8 }

```

```

{
  "data": {
    "reviews": [
      {
        "id": "5cdf8a929a583a6bf62ad5a9",
        "text": "Passing",
        "stars": 3,
        "fullReview": "Someone on the internet gave 3 stars, saying: \"Passing\""
      },
      {
        "id": "5cdf8abe3fc0886c2e3c4d36",
        "text": "Passing",
        "stars": 3,
        "fullReview": "Someone on the internet gave 3 stars, saying: \"Passing\""
      }
    ]
  }
}

```

## Custom scalars

If you're jumping in here, `git checkout 6_0.2.0` (tag `6_0.2.0`, or compare `6...7`)

In the last section we mentioned that the `ID` scalar is serialized like a string, but what does that process look like, and how do we make our own scalars? The only built-in scalars are `Int`, `Float`, `String`, `Boolean`, and `ID`. Another scalar type that most apps use is a date. For example, it would be nice to have a `Review.createdAt`. We could make it an `Int`, but then is it seconds or milliseconds since the [Unix epoch](#)? Or it could be a `String`, but there are a lot of string date formats out there. And both ways are missing validation (testing whether the string is a valid date string) and the improved understanding that comes from being able to know, looking at the schema, which fields are meant to be dates. So let's make our own `Date` scalar. We can add it to our schema:

`src/schema/schema.graphql`

```
scalar Date

type Query {
  hello: String!
  isoString(date: Date!): String!
}

#import 'Review.graphql'
#import 'User.graphql'
```

`src/schema/Review.graphql`

```
type Review {
  id: ID!
  text: String!
  stars: Int
  fullReview: String!
  createdAt: Date!
  updatedAt: Date!
}

...
```

First we declare the new scalar type ( `scalar Date` ), and then we use it for a new `isoString` query as well as `createdAt` and `updatedAt` fields on `Review`. We make them non-nullable because all `Review` objects will have them.

We can use the word `Date` for our type because we don't have other types of dates or times in our app. If we also had a `Date` that had no time component, like a birthday, or a `Time` that had no date component, like 14:00 (2 p.m.), we could call our new scalar `DateTime`.

`isoString` takes a `Date` as an argument and returns the date formatted as a string in the [ISO format](#):

`src/resolvers/index.js`

```
const resolvers = {
  Query: {
    hello: () => ' ',
    isoString: (_, { date }) => date.toISOString()
  }
}
```

Next we add to our resolvers a `GraphQLScalarType`, which tells Apollo Server how to handle a custom scalar. It will look like this:

`src/resolvers/Date.js`

```
import { GraphQLScalarType } from 'graphql'

export default {
  Date: new GraphQLScalarType({
    name:
```

```

    description:
      parseValue(value) {}
      parseLiteral(ast) {}
      serialize(date) {}
  })
}

```

`GraphQLScalarType` takes five parameters:

- `name` matches the scalar name we added to the schema, so `'Date'`
- `description` is shown in the schema section of GraphQL and Playground. It says what the scalar represents and how it appears in the JSON response from a server. The built-in description for `ID`, for instance, is:

The `ID` scalar type represents a unique identifier, often used to refetch an object or as a key for a cache. The ID type appears in a JSON response as a String; however, it is not intended to be human-readable. When expected as an input type, any string (such as `"4"`) or integer (such as `4`) input value will be accepted as an ID.

- `parseValue(value)` is a function called when the server receives a query variable for a `Date` argument. The variable's value is passed to `parseValue()`, and the function should return the value in our desired format—in this case, a JavaScript Date object. For example, if the client sends this query:

```

query ISOString($date: Date!) {
  isoString(date: $date)
}

```

with this as the variables JSON:

```

{
  "date": 1442188800000
}

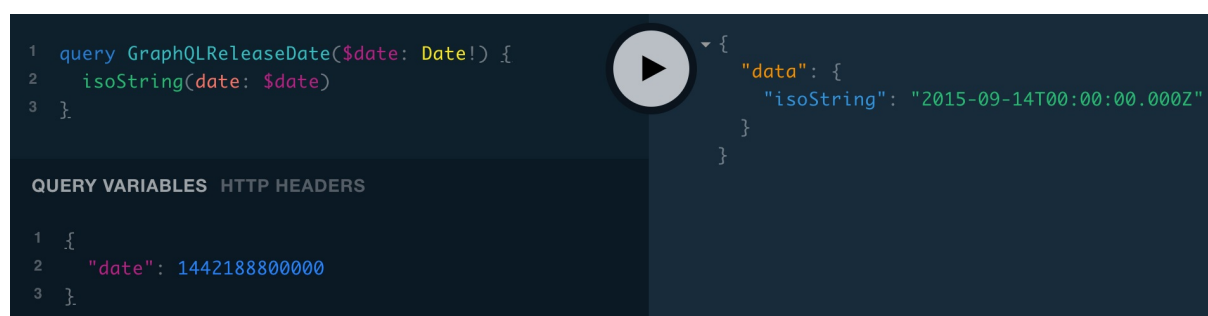
```

then `parseValue` is passed the integer `1442188800000` and should return a JS Date object, which Apollo Server will provide to our resolver, which calls `.toISOString()` on the JS Date object:

```

isoString: (_, { date }) => date.toISOString()

```



- `parseLiteral(ast)` is called when the server receives a query with a literal argument—meaning the argument is written in the query document itself instead of being provided separately in JSON (as variables are). `ast` stands for abstract syntax tree, which is an object that Apollo Server uses to parse the query document. `ast.value` has the literal value, and is always a string. Similar to `parseValue()`, `parseLiteral()` should return the server's internal representation of the scalar type. If the client sends this query document:

```

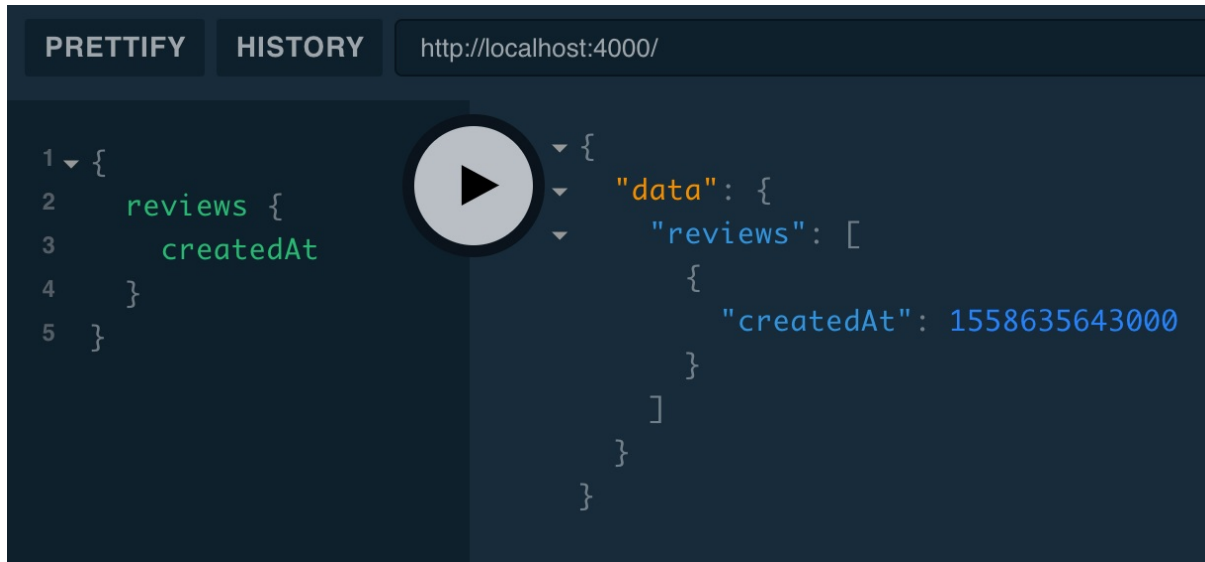
{
  isoString(date: 1442188800000)
}

```



Then `parseLiteral(ast)` will be called, and `ast.value` will be `"1442188800000"`.

- `serialize(date)` is called when the server is formatting a JSON response for the client. A resolver returns a JS Date object, then Apollo Server calls `serialize()` with that object, and `serialize()` returns the date in a format that can be put into the JSON response—which in our implementation of the `Date` scalar is an integer. For example, if the `Review.createdAt` resolver returns a JS Date, we would see an integer in the response:



If you're following along, this query won't work until we fill in `Date.js` and add it to `src/resolvers/index.js`.

Here's a basic implementation of the above:

```
import { GraphQLScalarType } from 'graphql'

export default {
  Date: new GraphQLScalarType({
    name: 'Date',
    description: `The \Date\ scalar type represents a single moment in time.
    It is serialized as an integer, equal to the number of milliseconds since
    the Unix epoch.`,
    parseValue: value => new Date(value),
    parseLiteral: ast => new Date(parseInt(ast.value)),
    serialize: date => date.getTime()
  })
}
```

`parseValue()` takes the integer and creates a `Date`. `parseLiteral()` gets the `ast.value` string, converts it into an integer, and creates a `Date`. `serialize()` takes the date and returns the milliseconds since epoch.

One important aspect of defining a custom scalar that we're missing is validation. If we check the values we're getting and throw errors with descriptive messages, it will help people using our API. Let's do that:

`src/resolvers/Date.js`

```
import { GraphQLScalarType } from 'graphql'
import { Kind } from 'graphql/language'

const isValid = date => !isNaN(date.getTime())

export default {
  Date: new GraphQLScalarType({
    name: 'Date',
    description: `The \Date\ scalar type represents a single moment in time. It is serialized as an integer, equal to the number of milliseconds since the Unix epoch.`,
    parseValue: value => {
      if (typeof value !== 'string') {
        throw new Error('Date: parseValue() expects a string')
      }
      const date = new Date(value)
      if (!isValid(date)) {
        throw new Error('Date: parseValue() expects a valid date')
      }
      return date
    },
    parseLiteral: ast => {
      if (ast.kind !== Kind.STRING) {
        throw new Error('Date: parseLiteral() expects a string')
      }
      const value = parseInt(ast.value)
      if (isNaN(value)) {
        throw new Error('Date: parseLiteral() expects a valid integer')
      }
      const date = new Date(value)
      if (!isValid(date)) {
        throw new Error('Date: parseLiteral() expects a valid date')
      }
      return date
    },
    serialize: date => {
      if (date instanceof Date) {
        return date.getTime()
      }
      throw new Error('Date: serialize() expects a Date object')
    }
  })
}
```

```

parseValue(value) {
  if (!Number.isInteger(value)) {
    throw new Error('Date values must be integers')
  }

  const date = new Date(value)
  if (!isValid(date)) {
    throw new Error('Invalid Date value')
  }

  return date
},

parseLiteral(ast) {
  if (ast.kind !== Kind.INT) {
    throw new Error('Date literals must be integers')
  }

  const date = new Date(parseInt(ast.value))
  if (!isValid(date)) {
    throw new Error('Invalid Date literal')
  }

  return date
},

serialize(date) {
  if (!(date instanceof Date)) {
    throw new Error(
      'Resolvers for Date scalars must return JavaScript Date objects'
    )
  }

  if (!isValid(date)) {
    throw new Error('Invalid Date scalar')
  }

  return date.getTime()
}
})
}

```

In `parseValue()` and `parseLiteral()`, we check whether the client sent an integer, then we create a JS Date and check whether it's valid. In `serialize()` we check that the value returned from a resolver is a JS Date object, then we check if it's a valid date, and finally we return the milliseconds since epoch.

We add this file to our resolvers in `resolvers/index.js` by importing and adding to our `resolversByType` array:

`src/resolvers/index.js`

```

...

import Review from './Review'
import User from './User'
import Date from './Date'

export default [resolvers, Review, User, Date]

```

We saw our `isoString` query working above, but now if we make a mistake, we get a helpful error message:

```

1 {
2   isoString(date: "9/14/2015")
3 }

```

```

{
  "error": {
    "errors": [
      {
        "message": "Expected type Date!, found \"9/14/2015\"; Date literals must be integers",
        "locations": [
          {
            "line": 2,
            "column": 19
          }
        ]
      }
    ]
  }
}

```

```

1 query GraphQLReleaseDate($date: Date!) {
2   isoString(date: $date)
3 }

```

QUERY VARIABLES HTTP HEADERS

```

1 {
2   "date": "foo"
3 }

```

```

{
  "error": {
    "errors": [
      {
        "message": "Variable \"$date\" got invalid value \"foo\"; Expected type Date; Date values must be integers",
        "locations": [
          {
            "line": 1,
            "column": 19
          }
        ]
      }
    ]
  }
}

```

The last part of our schema change for which we have to implement resolvers is `Review`'s `createdAt` and `updatedAt`. In MongoDB, the creation time is included in the default ID format, `ObjectId`. The first 4 bytes are the seconds since Unix epoch, so we can get the creation time from that. (And since it's the first 4 bytes, we can also sort by an `ObjectId` to order by most/least recently created.) The `mongodb` node library provides a method `ObjectId.getTimestamp()` that extracts the date for us:

`src/resolvers/Review.js`

```

export default {
  Query: ...
  Review: {
    ...
    createdAt: review => review._id.getTimestamp()
  },
  Mutation: ...
}

```

`updatedAt` is a field that we'll have to store in the database when reviews are created and update when reviews are modified. We don't have a way of modifying reviews yet, so we'll just add a line to our creation method:

`src/data-sources/Reviews.js`

```

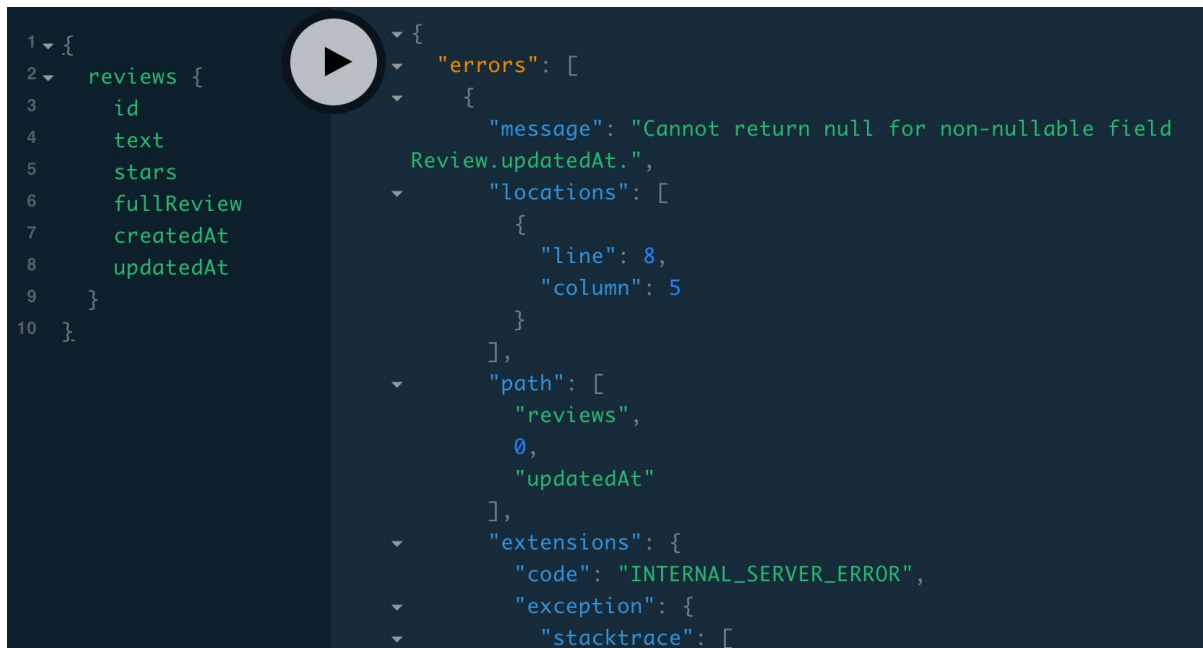
import { MongoDataSource } from 'apollo-datasource-mongodb'

export default class Reviews extends MongoDataSource {
  ...

  create(review) {
    review.updatedAt = new Date()
    this.collection.insertOne(review)
    return review
  }
}

```

Now we can include `updatedAt` in our `reviews` query, but we get the error `Cannot return null for non-nullable field Review.updatedAt :`



Apollo Server is telling us that it can't return `null` for `Review.updatedAt` to the client because the schema says it's a non-nullable field. Why is it *trying* to return `null` for `Review.updatedAt`? It's not—our resolver is. Our `reviews` resolver is returning reviews fetched from the database, but none of them have an `updatedAt` property because they were inserted before we updated our `Reviews.create()` data source method. We could fix our reviews in the database by adding an `updatedAt` field, but let's just delete them and re-create. If you'd like a GUI (*Graphical User Interface*, i.e., a program that runs in its own window instead of in the command line) for interacting with MongoDB, we recommend [MongoDB Compass](#). Here's how to delete all of our reviews using the `mongo` command-line shell:

```
$ mongo
MongoDB shell version v4.0.3
connecting to: mongoddb://127.0.0.1:27017
...

> use guide
switched to db guide
> db.reviews.find({})
{ "_id" : ObjectId("5cdfb1946df8548efb438535"), "text" : "Passing", "stars" : 3 }
{ "_id" : ObjectId("5cdfb1e4a1cf288f4d86dced"), "text" : "Passing", "stars" : 3 }
{ "_id" : ObjectId("5cdfb28e48435b90119bd2c6"), "text" : "Passing", "stars" : 3 }
> db.reviews.remove({})
WriteResult({ "nRemoved" : 3 })
> db.reviews.find({})
> exit
bye
```

Our second call to `db.reviews.find({})` doesn't show results because the collection is now empty. And when we do our `reviews` query, we get back an empty array. Now if we use Playground to send a `createReview` mutation, then we can do a `reviews` query with the `createdAt` and `updatedAt` fields:

The last three digits of `createdAt` will always be `000` because the API returns milliseconds since Epoch, and all that's stored in the `ObjectId` is *seconds* since Epoch.

An alternative to clearing the database collection would have been to add a resolver for `Review.updatedAt` that returns `Review.createdAt` when there's no `updatedAt` property on the review object. In order to call another resolver, we'd need to name the resolver's object and move `export default` to the end:

`src/resolvers/Review.js`

```
const resolvers = {
  Query: {
    reviews: ...
  },
  Review: {
    id: ...
    fullReview: ...
    createdAt: review => review._id.getTimestamp(),
    updatedAt: review => review.updatedAt || resolvers.Review.createdAt(review)
  },
  Mutation: {
    createReview: ...
  }
}

export default resolvers
```

Then we could reference another resolver function ( `resolvers.Review.createdAt(review)` ).

In this section we created a new `Date` scalar type, added `Query.isoString`, which has a `Date` argument, and `Review.createdAt` and `Review.updatedAt`, which resolve to `Date`s. We'll continue to use the `Date` type in the rest of our app, for instance for `User.createdAt/updatedAt` in the next section.

## Creating users

- [Protecting with secret key](#)
- [Setting user context](#)
- [Linking users to reviews](#)

Currently our `User` type just has two fields ( `firstName` and `lastName` ), and we aren't storing users in the database. If we wanted to continue without storing users in the database, we could fetch any further information we want, like email address or GitHub username, from Auth0 or GitHub whenever we needed it. However, this would be a little more complicated than querying our database, introduce latency (it takes longer for our server to talk to their servers than to query our database), and introduce another point of failure (if their services went down or there was a network failure between us and them). Furthermore, we're going to have to store some new user data (for instance, which sections they've read, or which reviews they've favorited), so we might as well have other user data we need stored along with it. In the first part of this section, we'll create user documents in a new `users` Mongo collection. In the second part, we'll query the collection to set the user context for resolvers.

## Protecting with secret key

If you're jumping in here, `git checkout 7_0.2.0` (tag `7_0.2.0`, or compare `7...8`)

There are two ways we could create our user doc. One is, in our context function, checking if the user we decode from the JWT exists in the database, and if they don't, fetching their data from Auth0 and GitHub and saving it to the database. The other method is to use an Auth0 hook—a function we write that runs on a certain trigger. The “Post User Registration” hook runs whenever a user first uses their GitHub account to log in. Inside of our hook function, we can put together the user data we want and send it to the server in a mutation. The Guide hook looks something like this:

```
const request = require('graphql-request').request
const pick = require('lodash').pick
```

```
const query = `
mutation createUserFromHook($user: CreateUserInput!, $secretKey: String!) {
  createUser(user: $user, secretKey: $secretKey) {
    id
  }
}
`

module.exports = function (user, context, cb) {
  const secretKey = context.webtask.data.secretKey
  const input = pick(user, 'username', 'email')
  input.authId = user.id
  const variables = {
    user: input,
    secretKey
  }
  request('https://api.graphql.guide/graphql', query, variables).then(data => cb(null, data))
};
```

The exported function is given data about the user, and then sends a `createUser` mutation to the Guide server. The mutation takes as arguments both the user data and a `secretKey` —a secret string that the server verifies before running the mutation, so that no one but the hook can create users.

When we want to protect a query, mutation, or field from being accessed by anyone, normally we use a JWT in the authorization header. We could create a JWT for this purpose, but it's easier to generate a random string (i.e. key). We could put the key in the authorization header like is usually done for API keys, which would look like this:

`src/context.js`

```
import { getAuthIdFromJWT } from './util/auth'

const API_KEYS = ['alohomora', 'speak-friend']

export default async ({ req }) => {
  const context = {}

  if (API_KEYS.includes(req.headers.authorization)) {
    context.apiUser = true
  } else {
    const jwt = req.headers.authorization
    const authId = await getAuthIdFromJWT(jwt)
    if (authId === 'github|1615') {
      context.user = {
        firstName: 'John',
        lastName: 'Resig'
      }
    }
  }

  return context
}
```

We add an if statement and set `context.apiUser` to `true`, which we can check inside our resolvers.

However, since we only need the key for this one mutation, we'll add a `secretKey` argument to it. As always, we start with the schema:

`src/schema/User.graphql`

```
type User {
  firstName: String!
  lastName: String!
}

extend type Query {
  me: User!
}
```

```

extend type Mutation {
  createUser(user: CreateUserInput!, secretKey: String!): User
}

input CreateUserInput {
  firstName: String!
  lastName: String!
  username: String!
  email: String!
  authId: String!
}

```

We're extending the `Mutation` type that first appears in `src/schema/Review.graphql`, and we follow the standard practice of our creation mutation resolving to the type it creates, `User`. And we create a new input type with the user fields we want. Next, we implement the `createUser` resolver:

`src/resolvers/User.js`

```

export default {
  Query: {
    me: (_, __, context) => context.user
  },
  Mutation: {
    createUser(_, { user, secretKey }, context) {
      // TODO
    }
  }
}

```

We have three things to do in our resolver:

- verify `secretKey` is correct
- create the user
- return the user

Best practice is to avoid committing secrets to git, so we won't do `if (secretKey !== 'foo')`. Instead, we'll use the `dotenv` package to set an environment variable. First we need to generate a secret:

```

$ node
> require('crypto').randomBytes(15, (e, buffer) => console.log(buffer.toString('hex')))
9e769699fae6f594beafb46e9078c2
> .exit

```

Then we put it in a file named `.env`:

```
SECRET_KEY=9e769699fae6f594beafb46e9078c2
```

That we have git ignore:

`.gitignore`

```

node_modules/
dist/
.env

```

And then we have `dotenv` read the values listed in `.env` into `process.env` at the beginning of our code (the first line of `src/index.js`):

```

import 'dotenv/config'
import { ApolloServer } from 'apollo-server'
import typeDefs from './schema/schema.graphql'
...

```

And then we can reference `process.env.SECRET_KEY` in our code:

`src/resolvers/User.js`

```
import { AuthenticationError } from 'apollo-server'

export default {
  Query: ...
  Mutation: {
    createUser(_, { user, secretKey }, context) {
      if (secretKey !== process.env.SECRET_KEY) {
        throw new AuthenticationError('wrong secretKey')
      }

      // TODO
    }
  }
}
```

We'll learn about errors in the [Errors section](#).

The next step is creating the user, for which we need a users data source! We create a new file:

`src/data-sources/Users.js`

```
import { MongoDataSource } from 'apollo-datasource-mongodb'

export default class Users extends MongoDataSource {
  create(user) {
    user.updatedAt = new Date()
    this.collection.insertOne(user)
    return user
  }
}
```

The `create()` method adds an `updatedAt` property, inserts, and returns, just like our `Reviews` data source. We include our new data source in the index file:

`src/data-sources/index.js`

```
import Reviews from './Reviews'
import Users from './Users'
import { db } from '../db'

export default () => ({
  reviews: new Reviews(db.collection('reviews')),
  users: new Users(db.collection('users'))
})
```

So now `users` will be available in our resolvers at `context.dataSources.users` :

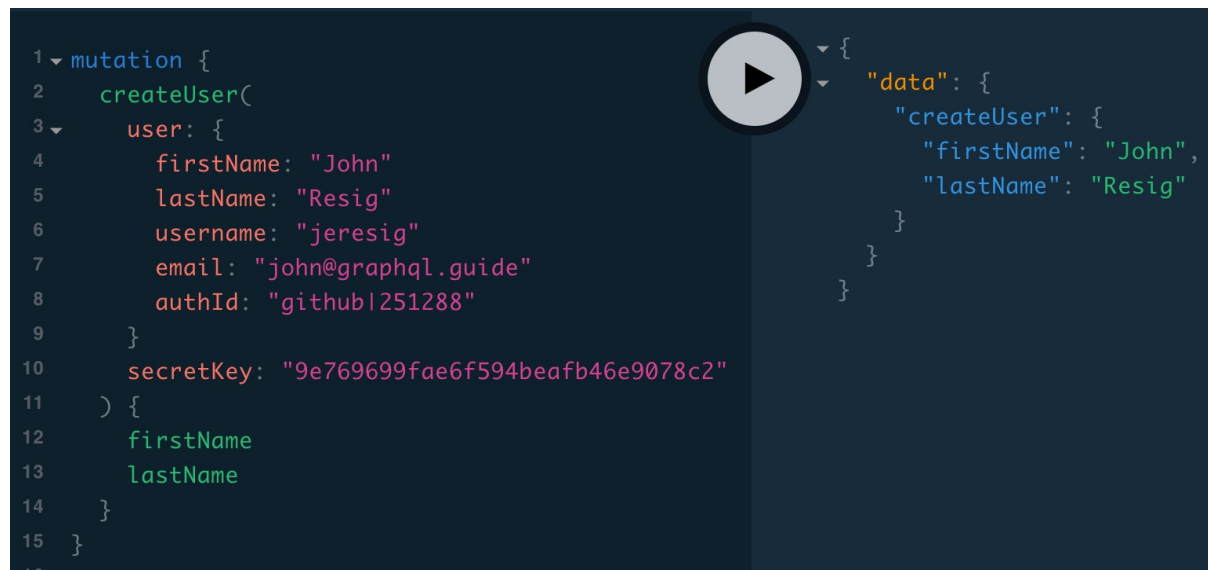
`src/resolvers/User.js`

```
export default {
  Query: ...
  Mutation: {
    createUser(_, { user, secretKey }, { dataSources }) {
      if (secretKey !== process.env.SECRET_KEY) {
        throw new AuthenticationError('wrong secretKey')
      }

      return dataSources.users.create(user)
    }
  }
}
```



Now the `createUser` should work (using your own data and `authId` for the `user` argument):



```

1 mutation {
2   createUser(
3     user: {
4       firstName: "John"
5       lastName: "Resig"
6       username: "jeresig"
7       email: "john@graphql.guide"
8       authId: "github|251288"
9     }
10    secretKey: "9e769699fae6f594beafb46e9078c2"
11  ) {
12    firstName
13    lastName
14  }
15 }

```

```

{
  "data": {
    "createUser": {
      "firstName": "John",
      "lastName": "Resig"
    }
  }
}

```

```

mutation {
  createUser(
    user: {
      firstName: "John"
      lastName: "Resig"
      username: "jeresig"
      email: "john@graphql.guide"
      authId: "github|1615"
    }
    secretKey: "9e769699fae6f594beafb46e9078c2"
  ) {
    firstName
    lastName
  }
}

```

## Setting user context

If you're jumping in here, `git checkout 8_0.2.0` (tag [8\\_0.2.0](#), or compare [8...9](#))

Now that we have our user document in the database, we can fetch it and put it in context:

```

src/context.js

import { getAuthIdFromJWT } from '../util/auth'
import { db } from './db'

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  const user = await db.collection('users').findOne({ authId })
  if (user) {
    context.user = user
  }

  return context
}

```

One possible concern with this method is latency—every authenticated request now has to wait for a round trip to the database before resolvers are run, and if the request is one that doesn't use `context.user`, we've wasted that time. It's usually not a long enough period of time to be concerned about, but if we were, we could solve it in a couple of

ways:

- Store whatever user data we needed in the JWT. Then we wouldn't have to fetch it from the database—we'd just decode it. This takes some additional coding, and what the code looks like depends on how you're creating the JWT (in this case we'd be talking to Auth0 via their API). JWTs have a limited size (~7k sent in an HTTP header), but that wouldn't be a limiting factor for us, since we don't have that much user data.
- Put a Promise on the context instead of the doc:

```
import { getAuthIdFromJWT } from './util/auth'
import { db } from './db'

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  context.userPromise = db.collection('users').findOne({ authId })

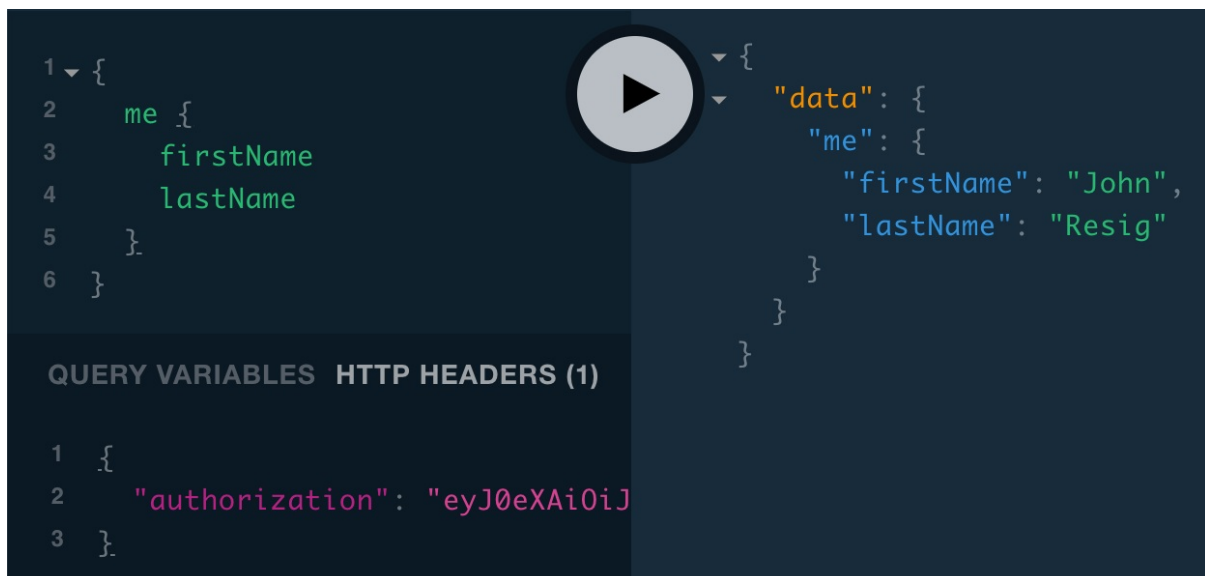
  return context
}
```

And then any resolvers that needed user data would do:

```
const user = await context.userPromise
```

That would clutter the code a little, so let's stick with our `context.user` code. ✨😊

Now if we do the `me` query (and set our authorization header as we did in the [Authenticating](#) section), we should be able to get the name from our user document:



There's more data about a user that our web client will need, so let's add to our schema:

`src/schema/User.graphql`

```
type User {
  id: ID!
  firstName: String!
  lastName: String!
  username: String!
  email: String!
  photo: String!
  createdAt: Date!
  updatedAt: Date!
}
```

```

}

extend type Query {
  me: User
}

extend type Mutation {
  createUser(user: CreateUserInput!, secretKey: String!): User
}

input CreateUserInput {
  firstName: String!
  lastName: String!
  username: String!
  email: String!
  authId: String!
}

```

`username`, `email`, and `updatedAt` are fields of the user document, so we don't need resolvers for them. We do need resolvers for `id`, `photo`, and `createdAt`. Also note that we don't have a `User.authId` field: while it's part of `CreateUserInput` and is stored in the user document, we don't need the client to be able to access it, so leaving it out of the `User` type means they won't be able to query for it.

For the `createdAt` resolver, we can do the same as the `Review.createdAt` resolver, calling the `getTimestamp()` method of the `ObjectId`:

`src/resolvers/User.js`

```

export default {
  Query: {
    me: (_, __, context) => context.user
  },
  User: {
    id: ({ _id }) => _id,
    photo(user) {
      // user.authId: 'github|1615'
      const githubId = user.authId.split('|')[1]
      return `https://avatars.githubusercontent.com/u/${githubId}`
    },
    createdAt: user => user._id.getTimestamp()
  },
  Mutation: ...
}

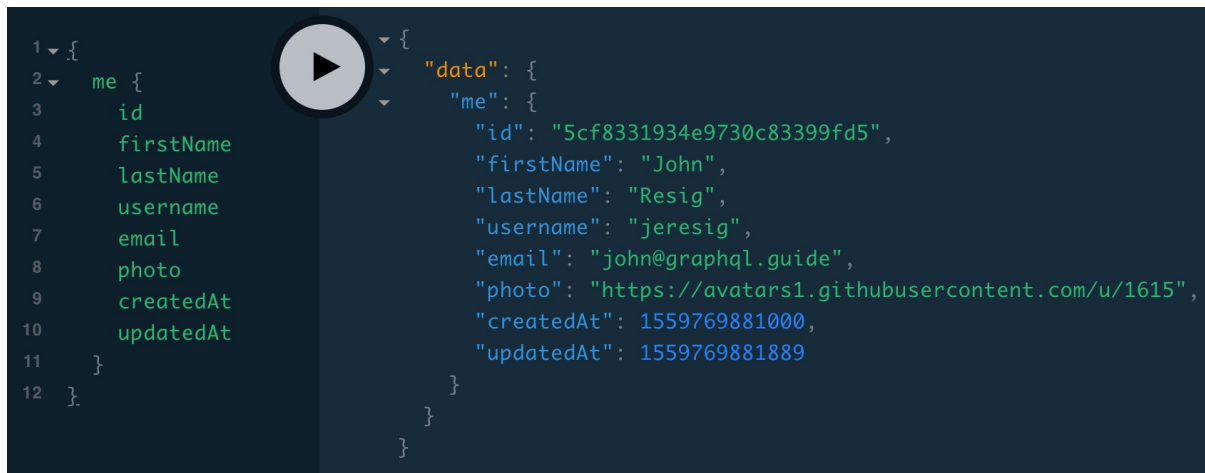
```

For the user's photo field, we can use GitHub avatar URLs, which have the GitHub user ID at the end, like:

`https://avatars.githubusercontent.com/u/1615`

And we can get the GitHub user ID number from the second part of the `authId`, after the `|` character (for example `github|1615`).

Now we can query for all `User` fields:



## Linking users to reviews

If you're jumping in here, `git checkout 9_0.2.0` (tag `9_0.2.0`, or compare `9...10`)

Another thing we can add now that we have a `users` collection is associate users with reviews. We want our client to be able to show the user's name and photo next to reviews, so we can update our `Review` type with an `author` field that resolves to a `User`:

`src/schema/Review.graphql`

```
type Review {
  id: ID!
  author: User!
  text: String!
  stars: Int
  fullReview: String!
  createdAt: Date!
  updatedAt: Date!
}
```

When we create the review, we need to save the author's ID. The author is the currently logged-in user, which is stored at `context.user`. Inside data sources, the context is available at `this.context`. So we can save `this.context.user._id` to an `authorId` field of the review document:

`src/data-sources/Reviews.js`

```
export default class Reviews extends MongoDataSource {
  ...

  create(review) {
    review.authorId = this.context.user._id
    review.updatedAt = new Date()
    this.collection.insertOne(review)
    return review
  }
}
```

Now our new `Review.author` resolver can use this `authorId` prop to fetch the user doc:

`src/resolvers/Review.js`

```
export default {
  Query: ...,
  Review: {
    id: ...,
    author: (review, _, { dataSources }) =>
      dataSources.users.findOneById(review.authorId),
  },
}
```

```

    fullReview: ...
    createdAt: ...
  },
  Mutation: ...
}

```

The next task is updating our current reviews in the database to have an `authorId` field (because we made `author` non-nullable, we'll get an error without one). Using our own user ID (from a `{ me { id } }` query) in the below

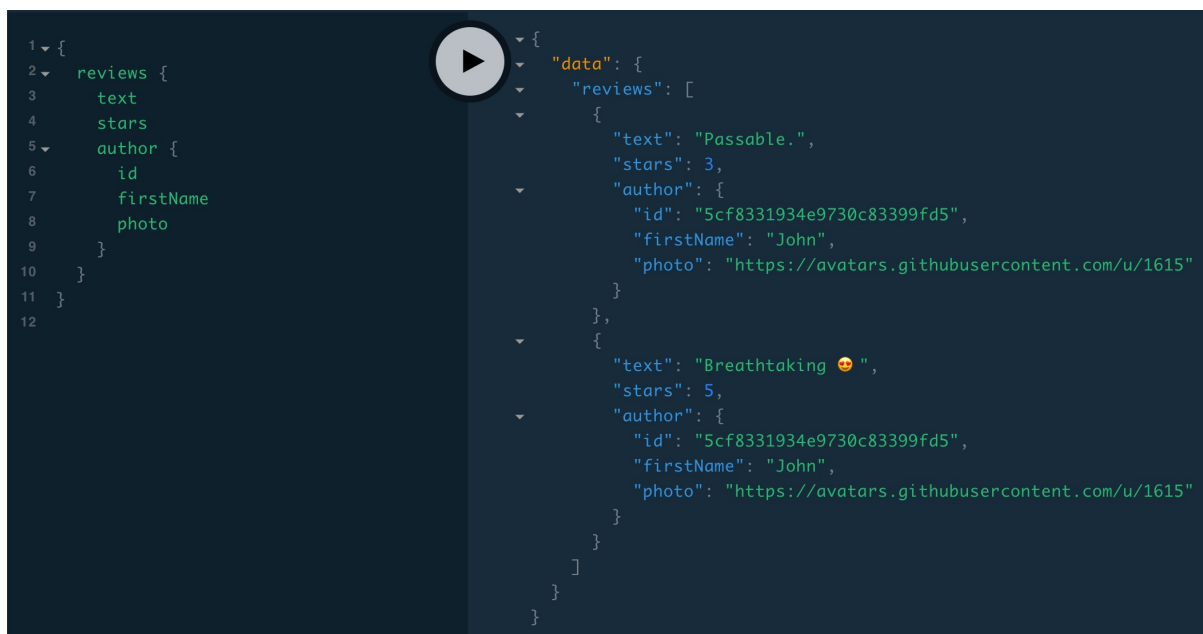
`ObjectId` :

```

$ mongo
> use guide
switched to db guide
> db.reviews.updateMany({}, {$set: {authorId: ObjectId('5cf8331934e9730c83399fd5')}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
> exit

```

we should now be able to add `author` to our selection set for our `reviews` query:

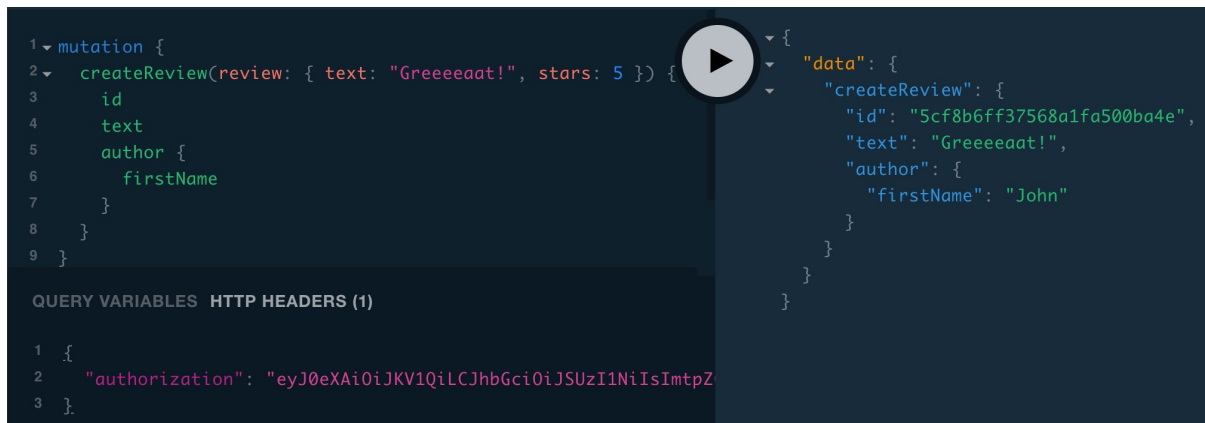


```

{
  reviews {
    text
    stars
    author {
      id
      firstName
      photo
    }
  }
}

```

And we should also be able to create a review and select the author, if we include our JWT in the authorization header:



The last thing to update is `Review.fullReview` : let's change "Someone on the internet gave N stars" to use the author's name. Currently we have:

src/resolvers/Review.js

```
export default {
  Query: {
    reviews: (_, __, { dataSources }) => dataSources.reviews.all()
  },
  Review: {
    id: review => review._id,
    author: (review, _, { dataSources }) =>
      dataSources.users.findOneById(review.authorId),
    fullReview: review =>
      `Someone on the internet gave ${review.stars} stars, saying: "${
        review.text
      }"`,
    createdAt: review => review._id.getTimestamp()
  },
  Mutation: ...
}
```

We'd like to do:

```
fullReview: review =>
  `${review.author.firstName} ${review.author.lastName} gave ${
    review.stars
  } stars, saying: "${review.text}"`,
```

But trying to query `{ reviews { fullReview } }` gives the error `Cannot read property 'firstName' of undefined`, which means that `review.author` is undefined. This is because `review` is a MongoDB document and has an `authorId` property, not an `author` property. We could either call the other resolver (as we saw in [Custom scalars](#) with `Review.updatedAt` ) or use the data source directly:

```
export default {
  Query: ...
  Review: {
    id: review => review._id,
    author: (review, _, { dataSources }) =>
      dataSources.users.findOneById(review.authorId),
    fullReview: async (review, _, { dataSources }) => {
      const author = await dataSources.users.findOneById(review.authorId)
      return `${author.firstName} ${author.lastName} gave ${
        review.stars
      } stars, saying: "${review.text}"`
    },
    createdAt: review => review._id.getTimestamp()
  },
  Mutation: ...
}
```

```
{
  reviews {
    fullReview
  }
}
```

```
1 {
2   reviews {
3     fullReview
4   }
5 }
6
```

```
{
  "data": {
    "reviews": [
      {
        "fullReview": "John Resig gave 3 stars, saying: \"Passable\""
      },
      {
        "fullReview": "John Resig gave 5 stars, saying: \"Breathtaking 🤩\""
      },
      {
        "fullReview": "John Resig gave 1 stars, saying: \"tldr\""
      }
    ]
  }
}
```

## Authorizing

If you're jumping in here, `git checkout 10_0.2.0` (tag `10_0.2.0`, or compare `10...11`)

In this section we'll implement an authorization check for a field on the `User` type. Later, in the [Error checking](#) section, we'll talk about how to find the places we need to do authorization checks.

Let's first add a new `user` query for fetching a single user by id:

`src/schema/User.graphql`

```
extend type Query {
  me: User
  user(id: ID!): User
}
```

`src/resolvers/User.js`

```
import { ObjectId } from 'mongodb'

export default {
  Query: {
    me: (_, __, context) => context.user,
    user: (_, { id }, { dataSources }) =>
      dataSources.users.findOneById(ObjectId(id))
  },
  User: ...,
  Mutation: ...
}
```

We have to turn the `id` string we receive as an argument into an `ObjectId` before calling `findOneById()`. The alternative would be to create an `objID` [custom scalar](#) that parsed string arguments into `ObjectId` objects, and then if we changed the argument type from `ID` to `objID`, then the `id` argument would be an `ObjectId` object by the time it reached our resolver, and we could call `findOneById()` directly:

```
extend type Query {
  me: User
  user(id: ObjID!): User
}
```

```
user: (_, { id }, { dataSources }) =>
  dataSources.users.findOneById(id)
```

```
import { GraphQLScalarType } from 'graphql'
import { ObjectId } from 'mongodb'

export default {
  ObjID: new GraphQLScalarType({
    name: 'ObjID',
    description: ...
    parseValue: value => ObjectId(value),
    parseLiteral: ast => ObjectId(ast.value),
    serialize: objectId => objectId.toString()
  })
}
```

Let's try our new `user` query:

```
1 {
2   user(id: "5cf8331934e9730c83399fd5") {
3     id
4     firstName
5     lastName
6     username
7     email
8     photo
9     createdAt
10    updatedAt
11  }
12 }
```

```
{
  "data": {
    "user": {
      "id": "5cf8331934e9730c83399fd5",
      "firstName": "John",
      "lastName": "Resig",
      "username": "jeresig",
      "email": "john@graphql.guide",
      "photo": "https://avatars.githubusercontent.com/u/1615",
      "createdAt": 1559769881000,
      "updatedAt": 1559769881889
    }
  }
}
```

We might now notice an issue. This query works without being logged in (i.e., including an authorization header), and it returns the user's email address. Similarly, we can query `{ reviews { author { email } } }` without being logged in. Our users would probably prefer their email addresses to not be publicly available! 😬

There are a few possible ways to solve this issue:

1. We could remove the `email` field from the `User` type. However, it would be nice to be able to show users their own email address on their profile page.
2. We could check whether the user is fetching their own email.

We could do the check in three places:

- **Resolver:** we just add an if statement to the beginning of a `User.email` resolver function.
- **Data source:** this doesn't have the granularity of the `User.email` resolver. If we threw an error in the data source method, the client wouldn't get any of the user's data. Doing authorization checks in data sources works well for preventing access to whole objects: for instance, if we wanted to prevent clients from fetching any user but their own. It works particularly well when there are multiple places in the schema the user can be accessed from. Instead of doing the check both in `Query.user` and `Review.author`, we can do it once in the `findOneById()` method of the `Users` data source.
- **Schema:** we can add a [custom directive](#) like `@isCurrentUser`:

```
type User {
  id: ID!
  firstName: String!
  lastName: String!
  email: String! @isCurrentUser
  ...
}
```



(And we'd make more directives for other authorization checks, like `@isLoggedIn` to deny access to a field from anonymous clients or `@isAdmin` to only allow admins to access a field.)

Wherever we do the check, when the user being requested doesn't match the logged-in user, we could either:

- Throw an error.
- Return `null`. The upside is it's easier for clients to handle than an error. (For example, if they query for 20 reviews with their authors, they'd get 20 errors to sort through.) The downside is they don't know why they're getting a `null` response—they might think the user just doesn't have an email.
- Use a union type that combines the normal result with the error result, like:

```
union EmailResult = Email | Forbidden

type Email {
  address: String!
  verified: Boolean!
}

type Forbidden {
  message: String!
}

type User {
  id: ID!
  firstName: String!
  lastName: String!
  email: EmailResult!
  ...
}
```

We'll cover [union errors](#) in the next section.

In this case, let's do the check in a resolver and throw an error. We currently don't have a resolver for `User.email`, because Apollo Server just uses the email property on the user object. It does the equivalent of this tiny resolver:

```
{
  User: {
    email: user => user.email
    ...
  }
}
```

When we provide our own resolver, Apollo Server will call our resolver instead of automatically returning `user.email`. Here's what our resolver looks like:

`src/resolvers/User.js`

```
import { ForbiddenError } from 'apollo-server'

export default {
  Query: {
    me: (_, __, context) => context.user,
    user: (_, { id }, { dataSources }) =>
      dataSources.users.findOneById(ObjectId(id))
  },
  User: {
    id: ({ _id }) => _id,
    email(user, _, { user: currentUser }) {
      if (!currentUser || !user._id.equals(currentUser._id)) {
        throw new ForbiddenError(`cannot access others' emails`)
      }

      return user.email
    },
    ...
  },
}
```

```

Mutation: ...
}

```

We'd have a naming conflict if we destructured `user` from context, so we assign to a new variable name `currentUser`. First we test whether there's any user at all, and then we test whether it's the same user. In the next section we'll see what the error looks like to the client! 🕒

## Errors

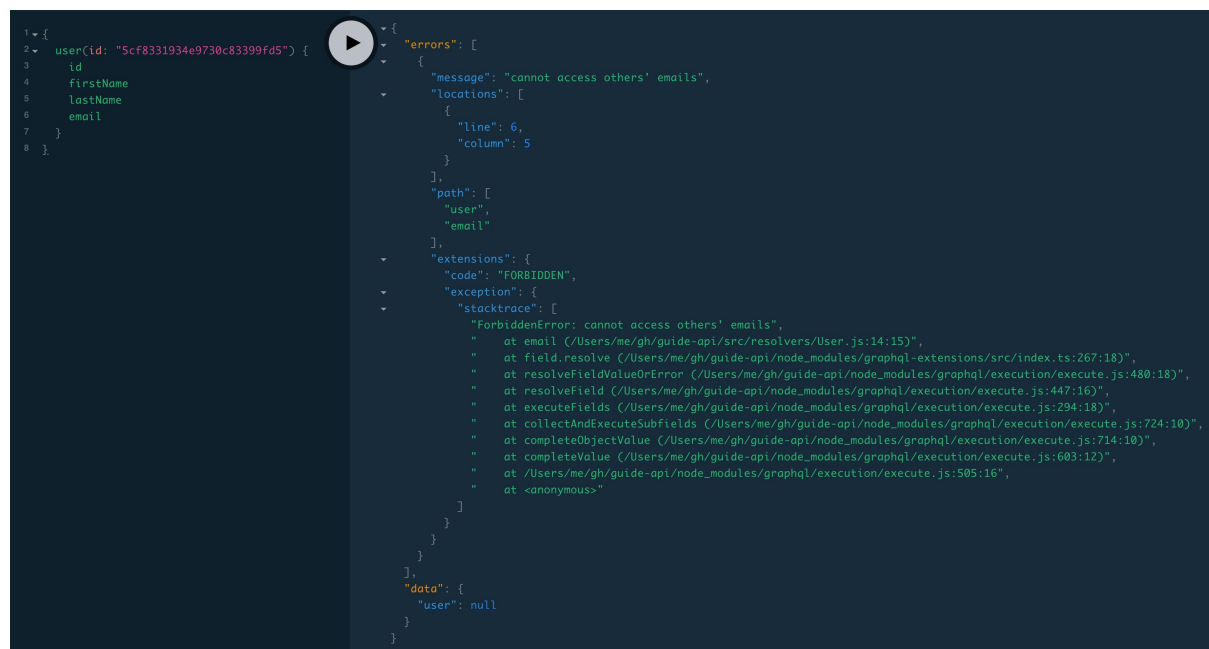
- [Nullability](#)
- [Union errors](#)
- [formatError](#)
  - [Logging errors](#)
  - [Masking errors](#)
- [Error checking](#)
- [Custom errors](#)

In [Nullability](#), we'll see what a thrown error looks like to the client, and we'll look at how data in the response changes based on whether fields are nullable. In [Union errors](#) we'll use the union type to return errors instead of throwing them. In [formatError](#) we log and mask errors, in [Error checking](#) we go through all the other errors we might want to check for or handle, and in [Custom errors](#) we create our own type of Apollo error.

## Nullability

If you're jumping in here, `git checkout 11_0.2.0` (tag `11_0.2.0`, or compare `11...12`)

In the last section, we throw an error when the client requests an email address and they're either not logged in or it's not their email. Let's see what that error looks like by making a `user` query without an authorization header:



```

1 {
2   user(id: "5cf8331934e9730c83399fd5") {
3     id
4     firstName
5     lastName
6     email
7   }
8 }

```

```

{
  "errors": [
    {
      "message": "cannot access others' emails",
      "locations": [
        {
          "line": 6,
          "column": 5
        }
      ],
      "path": [
        "user",
        "email"
      ],
      "extensions": {
        "code": "FORBIDDEN",
        "exception": {
          "stacktrace": [
            "ForbiddenError: cannot access others' emails",
            "    at email (/Users/me/gh/guide-api/src/resolvers/User.js:14:15)",
            "    at field.resolve (/Users/me/gh/guide-api/node_modules/graphql-extensions/src/index.ts:267:18)",
            "    at resolveFieldValueOrError (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:480:18)",
            "    at resolveField (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:447:16)",
            "    at executeFields (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:294:18)",
            "    at collectAndExecuteSubfields (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:724:10)",
            "    at completeObjectValue (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:714:10)",
            "    at completeValue (/Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:603:12)",
            "    at /Users/me/gh/guide-api/node_modules/graphql/execution/execute.js:505:16",
            "    at <anonymous>"
          ]
        }
      }
    }
  ],
  "data": {
    "user": null
  }
}

```

```

{
  user(id: "[id of a user in our database]") {
    id
    firstName
    lastName
    email
  }
}

```

```
}
}
```

We get an errors array with one item (an object with fields `message`, `locations`, `path`, and `extensions`) and `null` data:

```
{
  "errors": [
    {
      "message": "cannot access others' emails",
      "locations": [
        {
          "line": 6,
          "column": 5
        }
      ],
      "path": [
        "user",
        "email"
      ],
      "extensions": {
        "code": "FORBIDDEN",
        "exception": {
          "stacktrace": [
            "ForbiddenError: cannot access others' emails",
            ...
          ]
        }
      }
    }
  ],
  "data": {
    "user": null
  }
}
```

- The `message` matches the string we created our error with:

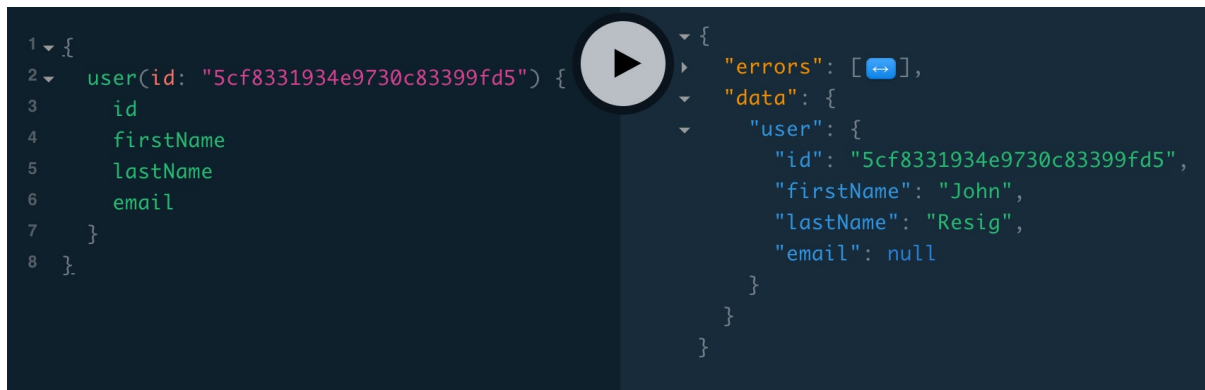
```
throw new ForbiddenError(`cannot access others' emails`)
```

- The `path` says the error occurred in the `email` field of the `user` query, and `locations` gives the line and column number of the `email` field in the client's query document.
- `extensions.code` is set to `FORBIDDEN` by the `ForbiddenError()` we're using. If we use a plain `Error` ( `throw new Error("cannot access others' emails")` ), then `extensions.code` would be `INTERNAL_SERVER_ERROR`.
- The stack trace is included unless `NODE_ENV` is set to `'production'`.

It would be nice if the server returned the rest of the user data we requested ( `id`, `firstName`, and `lastName` ) instead of just `null`. The reason it doesn't is `User.email` is non-nullable ( `String!` ), so a *null cascade* occurs: without an email value, the server isn't able to return a whole valid `User` type, so it returns `null` for the whole `query.user` field. If we make it nullable by removing the `!`, throwing an error from the `User.email` resolver will return `null` just for the `email` field—the server will still return the rest of the `User` fields:

```
src/schema/User.graphql
```

```
type User {
  id: ID!
  firstName: String!
  lastName: String!
  username: String!
  email: String
  ...
}
```



```

1 {
2   user(id: "5cf8331934e9730c83399fd5") {
3     id
4     firstName
5     lastName
6     email
7   }
8 }

```

```

{
  "errors": [],
  "data": {
    "user": {
      "id": "5cf8331934e9730c83399fd5",
      "firstName": "John",
      "lastName": "Resig",
      "email": null
    }
  }
}

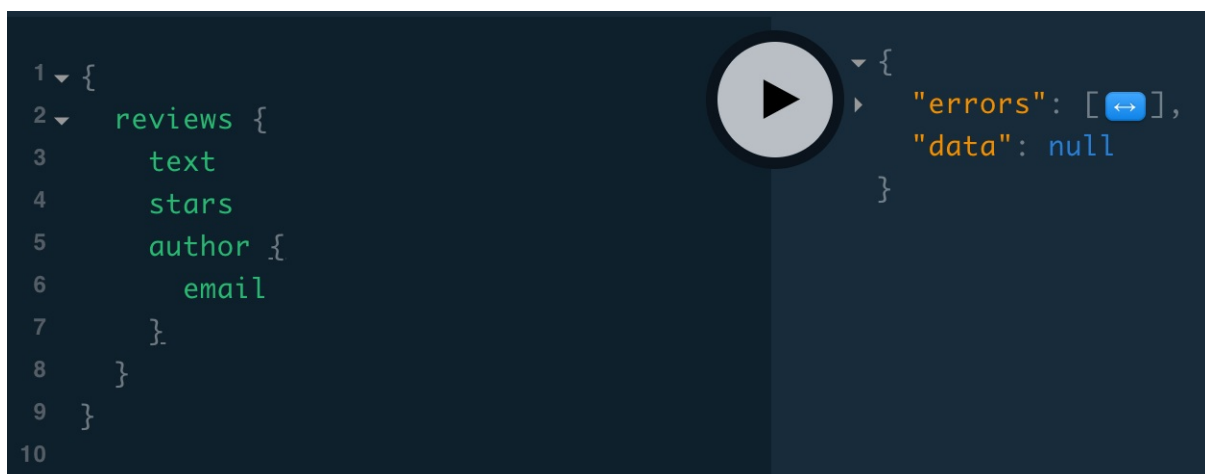
```

👤 This is a great improvement, especially since in the non-nullable case, a thrown error results in not just `null` for the user, but anything at a higher level as well! For example, here's a `reviews` query requesting a non-nullable email :

```

{
  reviews {
    text
    stars
    author {
      email
    }
  }
}

```



```

1 {
2   reviews {
3     text
4     stars
5     author {
6       email
7     }
8   }
9 }
10

```

```

{
  "errors": [{}],
  "data": null
}

```

Apollo server tries to return `null` for email, but it's non-nullable, so then it tries to return `null` for `Review.author`, but it's non-nullable, so then it tries to return `null` for the review, but the review is non-nullable and the list of reviews is non-nullable so we don't even end up with `"data": {"reviews": null}` —we just get `"data": null` !

So when we throw an error for a certain field but still want the client to get the rest of the data, we want to remember to make that field nullable. ❌ !

## Union errors

If you're jumping in here, `git checkout 12_0.2.0` (tag [12\\_0.2.0](#), or compare [12...13](#))

As mentioned in the [Authorizing](#) section, an alternative to throwing an error is returning `null`. The downside is the client can't determine whether the server is returning `null` because there's no data or because the client doesn't have access to it. It might be helpful to know they don't have access so that they can prompt the user to log in.

When a field's type is an object type, an alternative to returning `null` from the resolver is returning an error object. We can do this by changing the type to a union. Instead of:

```
type Query {
  item(id: Int!): Item
}

type Item {
  id: Int
  name: String
}
```

we can do:

```
type Query {
  item(id: Int!): ItemResult
}

type Item {
  id: Int
  name: String
}

type ItemError {
  reason: String
}

union ItemResult = Item | ItemError
```

Now the `item` query resolver is able to return either an `Item` or an `ItemError`. This query:

```
{
  item(id: 1) {
    __typename
    ... on Item {
      name
    }
    ... on ItemError {
      reason
    }
  }
}
```

can return either of these two JSON responses:

```
{
  "data": {
    "item": {
      "__typename": "Item",
      "id": 1,
      "name": "GraphQL hacky sack"
    }
  }
}
```

```
{
  "data": {
    "item": {
      "__typename": "ItemError",
      "reason": "This item has been discontinued."
    }
  }
}
```

Why do this? It can be easier for the client to handle the errors if they're inline in the `"data"` attribute of the JSON rather than the `"errors"` attribute. For example, imagine a `searchUsers` query that returned a long list of users. If we wanted the client to be able to show some information about deleted or suspended users, and we threw errors for each one, the client would have to go through an array of `"errors"` in the JSON response and match them up with holes in the `data.searchUsers` results. Further, they would have to be familiar with what type of errors are thrown and the format of the error data. Versus if we document in the schema the types of *expected* errors and return them from resolvers, clients know what data possibilities to expect, and they can smoothly iterate over just the `data.searchUsers` JSON array that they get.

*Expected* is highlighted because unexpected errors (like an unauthorized error or database failure) are usually kept as thrown errors, for the client to handle outside of its normal process of presenting expected data on the screen.

Let's implement this `searchUsers` query to see what it looks like. As usual, we'll start with the schema:

`src/schema/User.graphql`

```
extend type Query {
  me: User
  user(id: ID!): User
  searchUsers(term: String!): [UserResult!]!
}

type DeletedUser {
  username: String!
  deletedAt: Date!
}

type SuspendedUser {
  username: String!
  reason: String!
  daysLeft: Int!
}

union UserResult = User | DeletedUser | SuspendedUser
```

A `UserResult` union type can be either a `User`, `DeletedUser`, or `SuspendedUser`, each of which have a `__typename` and `username` but have different other fields. Let's implement the `searchUsers` resolver next:

`src/resolvers/User.js`

```
export default {
  Query: {
    me: ...
    user: ...
    searchUsers: (_, { term }, { dataSources }) =>
      dataSources.users.search(term)
  },
}
```

We take the search `term` parameter and pass it to a `search()` method, which will talk to the database:

`src/data-sources/Users.js`

```
export default class Users extends MongoDataSource {
  ...

  search(term) {
    return this.collection.find({ $text: { $search: term } }).toArray()
  }
}
```

`$text: { $search: term }` does a [text search](#) of the `users` collection. For it to work, we need to create a `text` index, which includes all the fields we want to search—in this case, the `name` and `username` fields. In MongoDB, we usually use the `collection.createIndex()` method, which checks if the index already exists, and creates it if not. It would be nice to put the command in the same file as our `search()` method so that it's easy to see which fields are being searched. One method we know will get called is the constructor, so we can put it there:

```
export default class Users extends MongoDataSource {
  constructor(collection) {
    super(collection)

    this.collection.createIndex({
      firstName: 'text',
      lastName: 'text',
      username: 'text'
    })
  }
  ...
}
```

We're currently instantiating this data source with:

```
new Users(db.collection('users'))
```

so in order to maintain that functionality, we need to take that object argument `collection` and pass it to `super()`.

A new `Users` object is created for every request, which is far more often than we need to be calling `createIndex()`—once at server startup would be sufficient—but the performance impact is miniscule, so we needn't worry about it until we're at Google scale 😊.

Now our `search()` method returns a list of users, but they're all normal users—we don't have any suspended or deleted users yet. Let's create three users in our database, all with the same first name so that they come up in a single search:

If you generated your own secret key, use that. It's located in your `.env` file.

```
mutation {
  createUser(
    user: {
      firstName: "John"
      lastName: "Resig"
      username: "jeresig"
      email: "john@graphql.guide"
      authId: "github|1615"
    }
    secretKey: "9e769699fae6f594beafb46e9078c2"
  ) {
    firstName
    lastName
  }
}
```

```
mutation {
  createUser(
    user: {
      firstName: "John"
      lastName: "Smith"
      username: "jsmith"
      email: "jsmith@example.com"
      authId: "github|1"
    }
    secretKey: "9e769699fae6f594beafb46e9078c2"
  ) {
```

```

    firstName
    lastName
  }
}

```

```

mutation {
  createUser(
    user: {
      firstName: "John"
      lastName: "Rest"
      username: "rest4eva"
      email: "rest4eva@example.com"
      authId: "github|2"
    }
    secretKey: "9e769699fae6f594beafb46e9078c2"
  ) {
    firstName
    lastName
  }
}

```

Now we can use the mongo shell to mark John Rest deleted and John Smith suspended:

```

$ mongo
> use guide
> db.users.updateOne({ username: 'rest4eva' }, { $set: { deletedAt: new Date() } })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.users.updateOne({ username: 'jsmith' }, { $set: { suspendedAt: new Date(), durationInDays: 300, reason: 'Terms of Service violation' } })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

```

Now let's go back to our code—our resolver returns a list of users:

```

this.collection.find({ $text: { $search: term } }).toArray()

```

But we don't want all of the returned objects to be of type `User`—then the client would get the user data of the deleted/suspended users and not know they were deleted/suspended. Whenever we use a union type, we need to tell Apollo which objects are of which type. For that we use a special resolver called `__resolveType`:

`src/resolvers/User.js`

```

export default {
  Query: {
    me: ...
    user: ...
    searchUsers: (_, { term }, { dataSources }) =>
      dataSources.users.search(term)
  },
  UserResult: {
    __resolveType: result => {
      if (result.deletedAt) {
        return 'DeletedUser'
      } else if (result.suspendedAt) {
        return 'SuspendedUser'
      } else {
        return 'User'
      }
    }
  },
},

```

Now when we return an object from a resolver that's supposed to return a `UserResult`, Apollo gives that object to `UserResult.__resolveType()`, which returns the type of the object. So now the server can't return the `firstName` of a deleted user, because it's not a field of `DeletedUser` in the schema.



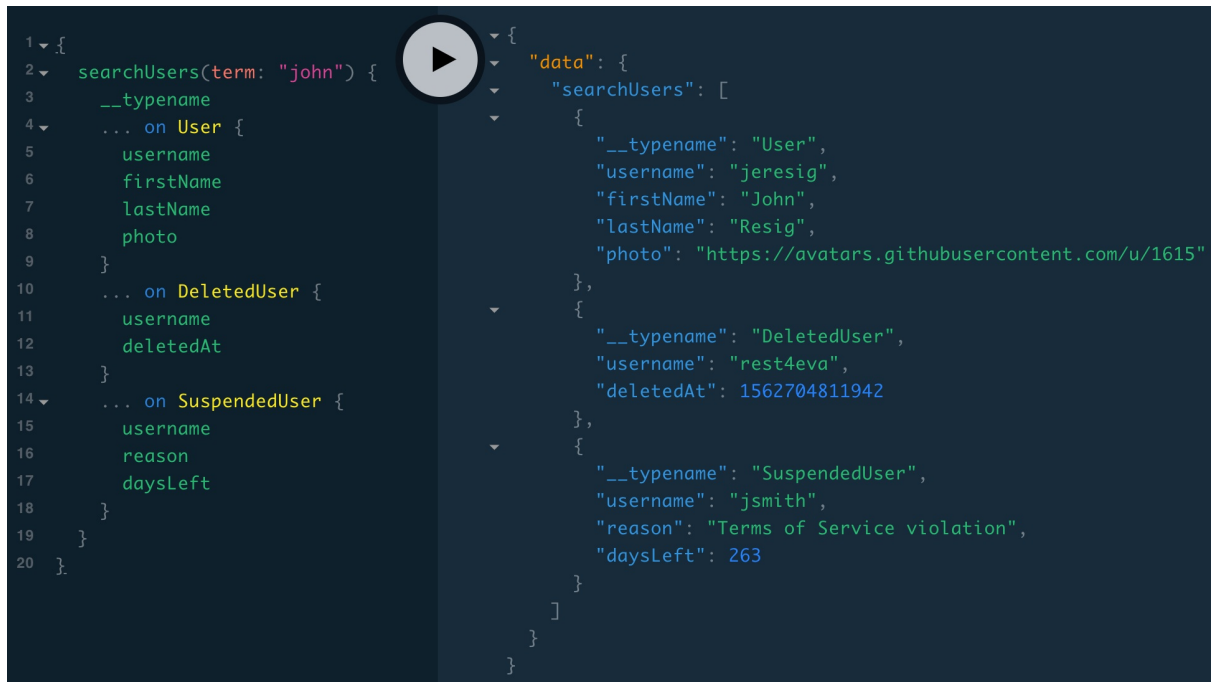
The last piece we need to add is `SuspendedUser.daysLeft`, which isn't stored in the database (we only store `suspendedAt` and `durationInDays` in the database). So we create a resolver for it:

```
import { addDays, differenceInDays } from 'date-fns'

export default {
  Query: ...,
  UserResult: ...,
  SuspendedUser: {
    daysLeft: user => {
      const end = addDays(user.suspendedAt, user.durationInDays)
      return differenceInDays(end, new Date())
    }
  },
}
```

`addDays` returns a date, and `differenceInDays` returns an integer. Now we can make our query:

```
{
  searchUsers(term: "john") {
    __typename
    ... on User {
      username
      firstName
      lastName
      photo
    }
    ... on DeletedUser {
      username
      deletedAt
    }
    ... on SuspendedUser {
      username
      reason
      daysLeft
    }
  }
}
```



Even though `username` is common to all possible types, with unions, the only field we can select outside of an `inline fragment` is the meta field `__typename`.

Now the client can iterate over `data.searchUsers` and check the `__typename`, and if it's a `DeletedUser` or `SuspendedUser`, display that user differently.

## formatError

There's an Apollo Server option called `formatError` that allows us to log and modify errors. In this section we'll see a couple situations in which we might use it.

## Logging errors

Background: [Json Web Tokens](#)

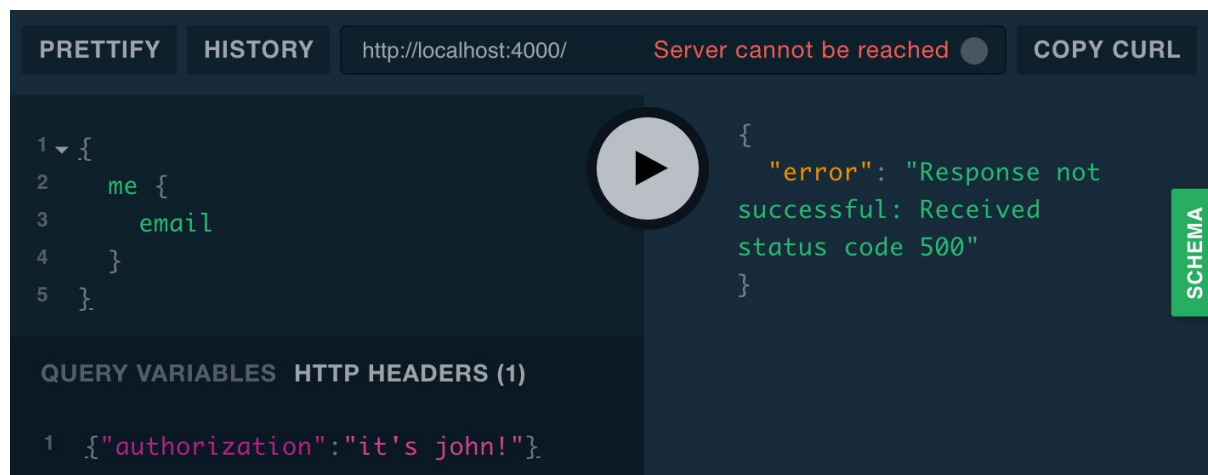
If you're jumping in here, `git checkout 13_0.2.0` (tag `13_0.2.0`, or compare `13...14`)

Usually when there are server errors, we see them in the `errors` field of the JSON response. In Playground, it's usually easy to see all the error information, including the stack trace, but when it's not easy to see the error on the client, it would be nice to be able to see the error in the server output on the command line. And in production we need some way of tracking the errors our users trigger.

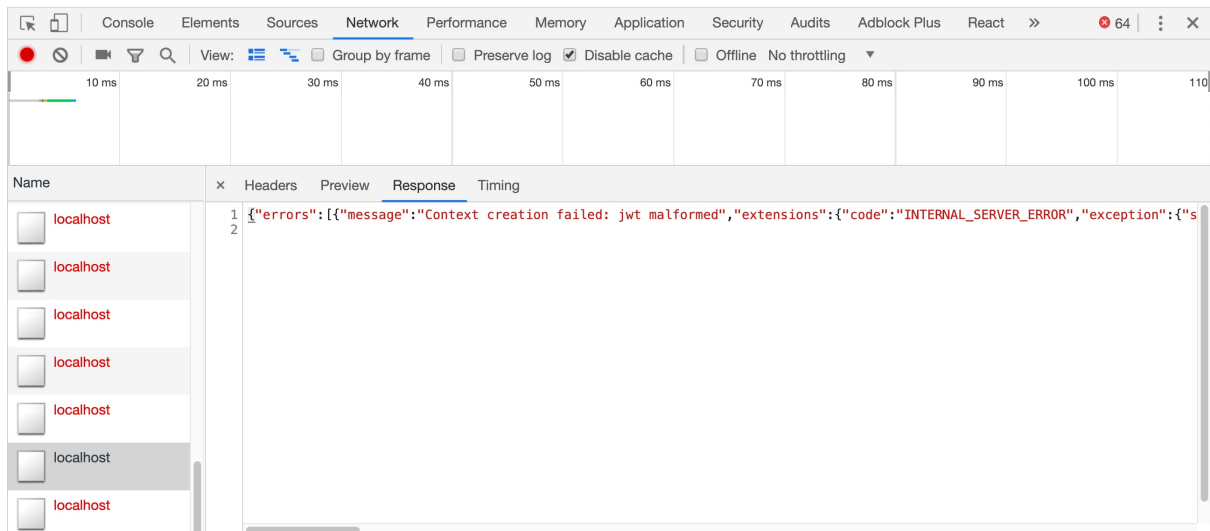
There is one case in which Playground doesn't conveniently show us the server error: when it receives an error from an introspection query. Playground periodically sends an introspection query to our server to get an up-to-date schema to back its query checking and schema tab. When we set an HTTP header, Playground uses it for the introspection query as well. So when we set an invalid authorization header, the server returns an error for the introspection query, but Playground might not show it to us—it might just say “Response not successful”:

```
{
  me {
    email
  }
}
```

```
{
  "authorization": "it's me, john!"
}
```



If we go into the devtools Network tab and select a `localhost` request, we can see the GraphQL `errors` field, but it's hard to read the stack trace—we either have to scroll and visually parse the newlines or paste it into a JSON formatter (we recommend `jq`: `brew install jq`, `copy`, `pbpaste | jq .`).



Since that takes a few steps, let's instead log the error using `formatError`:

`src/index.js`

```
import formatError from './formatError'

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError
})
```

`src/formatError.js`

```
export default error => {
  console.log(error)
  return error
}
```

Now the error is logged to the terminal:

```
{ [JsonWebTokenError: Context creation failed: jwt malformed]
  message: 'Context creation failed: jwt malformed',
  locations: undefined,
  path: undefined,
  extensions:
    { code: 'INTERNAL_SERVER_ERROR',
      exception: { stacktrace: [Array] } } }
```

But we don't see the stack trace, so let's log that as well, if the error has one:

```
import get from 'lodash/get'

export default error => {
  console.log(error)
  console.log(get(error, 'extensions.exception.stacktrace'))
  return error
}
```

And now we also get:

```
[ 'JsonWebTokenError: Context creation failed: jwt malformed',
```

```
'    at module.exports (/guide-api/node_modules/jsonwebtoken/verify.js:63:17)',
'    at internal/util.js:230:26',
'    at verify (/guide-api/src/util/auth.js:24:31)',
'    at ApolloServer._default [as context] (/guide-api/src/context.js:8:24)',
'    at ApolloServer.<anonymous> (/guide-api/node_modules/apollo-server-core/src/ApolloServer.ts:535:24)',
'    at Generator.next (<anonymous>)',
'    at /guide-api/node_modules/apollo-server-core/dist/ApolloServer.js:7:71',
'    at new Promise (<anonymous>)',
'    at __awaiter (/guide-api/node_modules/apollo-server-core/dist/ApolloServer.js:3:12)',
'    at ApolloServer.graphQLServerOptions (/guide-api/node_modules/apollo-server-core/dist/ApolloServer.js:31
6:16)' ]
```

And we can further debug! The error starts in `node_modules/ (/guide-api/node_modules/jsonwebtoken/verify.js:63:17)`, so let's look for the first lines that are inside our code (`src/`):

```
'    at verify (/guide-api/src/util/auth.js:24:31)',
'    at ApolloServer._default [as context] (/guide-api/src/context.js:8:24)',
```

Now let's look at `src/context.js`:

```
import { getAuthIdFromJWT } from './util/auth'
import { db } from './db'

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  const authId = await getAuthIdFromJWT(jwt)
  const user = await db.collection('users').findOne({ authId })
  if (user) {
    context.user = user
  }

  return context
}
```

Line 8 is `const authId = await getAuthIdFromJWT(jwt)`. So the error message `"jwt malformed"` means the authorization header is not formatted as a valid JWT.

We achieved our goal of using `formatError` to log the error so that we could debug it. We can't prevent clients from sending bad authorization headers, but we can improve the errors we throw. The two most common errors thrown during JWT parsing are `jwt malformed` and `jwt expired`, so let's cover those:

`src/context.js`

```
import { AuthenticationError } from 'apollo-server'

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  let authId

  if (jwt) {
    try {
      authId = await getAuthIdFromJWT(jwt)
    } catch (e) {
      let message
      if (e.message.includes('jwt expired')) {
        message = 'jwt expired'
      } else {
        message = 'malformed jwt in authorization header'
      }
      throw new AuthenticationError(message)
    }
  }

  const user = await db.collection('users').findOne({ authId })
```

```

    context.user = user
  }

  return context
}

```

We catch errors from `getAuthIdFromJWT()`, and use a different error message depending on the kind of error. Then we use Apollo's `AuthenticationError` error type, which adds an `extensions.code` of `"UNAUTHENTICATED"` to the error. The other errors that might occur are from the database (during the `findOne()`)—we'll cover these in the [next section](#). Let's also throw an error when there is no matching user in the database:

`src/context.js`

```

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization
  let authId

  if (jwt) {
    ...

    const user = await db.collection('users').findOne({ authId })
    if (user) {
      context.user = user
    } else {
      throw new AuthenticationError('no such user')
    }
  }

  return context
}

```

Now let's repeat our bad-header query and see what new error we get in the console:

```

{ [AuthenticationError: Context creation failed: malformed jwt in authorization header]
  message: 'Context creation failed: malformed jwt in authorization header',
  locations: undefined,
  path: undefined,
  extensions: { code: 'UNAUTHENTICATED', exception: { stacktrace: [Array] } } }
[ 'AuthenticationError: Context creation failed: malformed jwt in authorization header',
  '   at ApolloServer._default [as context] (/Users/me/gh/guide-api/src/context.js:21:13)',
  '   at <anonymous>',
  '   at runMicrotasksCallback (internal/process/next_tick.js:121:5)',
  '   at _combinedTickCallback (internal/process/next_tick.js:131:7)',
  '   at process._tickCallback (internal/process/next_tick.js:180:9)' ]

```

We now see the `UNAUTHENTICATED` error code and the more detailed error message. Our piece of the message — `malformed jwt in authorization header` —is preceded by `Context creation failed:`, which is added by Apollo for any errors that occur in the context function, and `AuthenticationError:`, which is taken from the name of the error object.

## Masking errors

If you're jumping in here, `git checkout 14_0.2.0` (tag `14_0.2.0`, or compare `14...15`)

`formatError()` isn't just for logging—as the name indicates, we can change the error. The most common change is masking an error we don't want the client to see.

You may have noticed that we return the error in the last line of the `formatError()` function:

`src/formatError.js`

```

export default error => {

```

```

console.log(error)
console.log(get(error, 'extensions.exception.stacktrace'))
return error
}

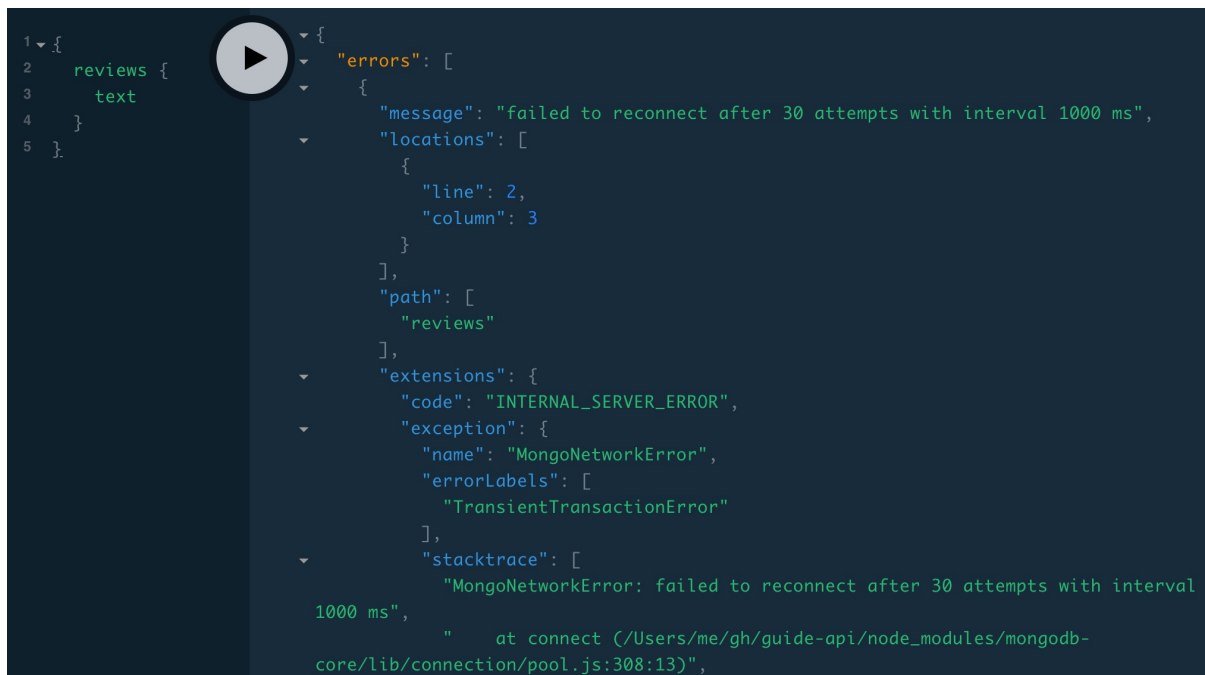
```

When an error is thrown in our code, Apollo catches it and gives it to `formatError()`, which returns an error object, which Apollo serializes into JSON and sends in the `errors` attribute to the client. Inside `formatError()`, we can modify the error object—by editing, adding, or removing properties—or return a new error.

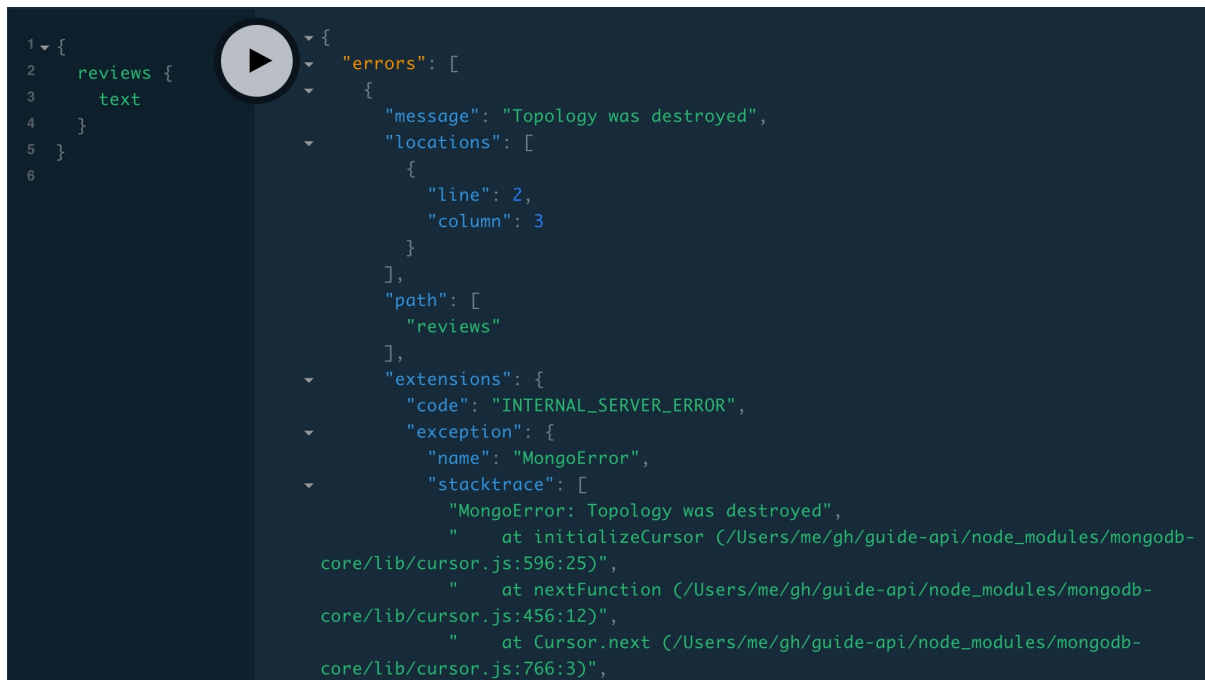
A common category of error to mask is database errors—we might want to hide the original error message for security reasons or to avoid confusing non-technical users with messages they don't understand. Let's see, for example, what errors happen when the server can't reach the database. We can stop the database with this command:

```
$ brew services stop mongodb-community
```

If we wait 30 seconds and then make a request, we get a `MongoNetworkError`:



And if we keep making requests, we start getting `"MongoError: Topology was destroyed"`:



Let's mask both of those with a new error:

```

export default error => {
  console.log(error)
  console.log(get(error, 'extensions.exception.stacktrace'))

  const name = get(error, 'extensions.exception.name') || ''
  if (name.startsWith('Mongo')) {
    return new Error('Internal server error')
  } else {
    return error
  }
}

```

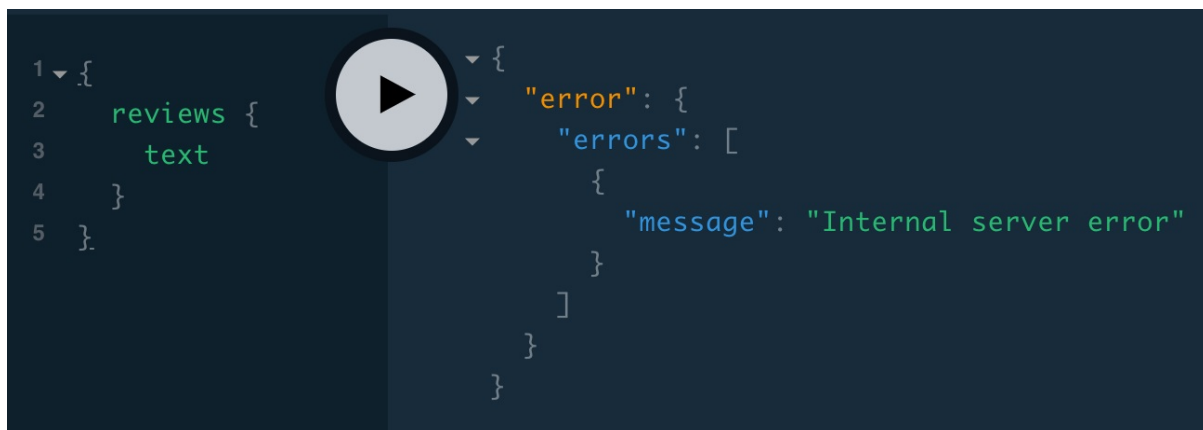
When we edit our code, the server fails to restart because it can't connect to the database. So in order to test, we can restart Mongo:

```
$ brew services start mongodb-community
```

And then restart the server, and then stop Mongo:

```
$ brew services stop mongodb-community
```

Now we get our masked error instead of either of the Mongo errors:



One last note on `formatError`—in production, we'll usually want to send our errors to an error tracking or logging service instead of logging them to the server console:

```
const inProduction = process.env.NODE_ENV === 'production'

export default error => {
  if (inProduction) {
    // send error to tracking service
  } else {
    console.log(error)
    console.log(get(error, 'extensions.exception.stacktrace'))
  }
  ...
}
```

## Error checking

If you're jumping in here, `git checkout 15_0.2.0` (tag `15_0.2.0`, or compare `15...16`)

So far we've dealt with the `User.email` authorization error, users who have been deleted or suspended, authentication errors, and MongoDB errors. Let's go through our entire app and think about all the possible errors we want to handle or throw:

- **Network:** If our node server is cut off from the internet, or if there's a DNS issue, the client won't be able to connect to our server, and will see an error that will look different depending on their browser or platform.
- **Servers**
  - **Node:** If our Node GraphQL application server isn't running, then the client won't be able to connect, and will see the same error as when there's a network failure.
  - **MongoDB:** We mask errors with our MongoDB server, including inability to connect, in `formatError()`
- **Request:** If the network request isn't a valid GraphQL HTTP request, then the error will be handled before it reaches our code—either by our server's operating system, Node, or Apollo Server.
- **Context:** Assuming the request is a valid GraphQL request (including valid against our schema), the server starts by setting the context for resolvers. This process often involves looking at request headers. We covered errors that might occur while creating context in the [Logging errors](#) section.
- **Resolvers:**
  - **Arguments:** Apollo validates the arguments' data types, but we often want to do further validation on the argument values.
  - **Execution:** We want to handle any possible errors that might occur in the running of our resolver code—things like invalid JWT decoding, dividing by zero, or trying to access a 3rd party service that's offline.
  - **Authorization:** If there's data or functions that we don't want certain people to access or trigger, we need to avoid returning the data / running the functions.



In this section we'll go through our resolvers. Let's start with authorization. For data access, let's look at our main data types:

```
type Review {
  id: ID!
  author: User!
  text: String!
  stars: Int
  fullReview: String!
  createdAt: Date!
  updatedAt: Date!
}

type User {
  id: ID!
  firstName: String!
  lastName: String!
  username: String!
  email: String
  photo: String!
  createdAt: Date!
  updatedAt: Date!
}
```

Depending on our app, we might consider `createdAt` and `updatedAt` to be sensitive, but for us, the only field we don't want to be public is `email`, which we already [have a check for](#). If we had an app for which an entire data type was restricted, then in order to verify it was restricted properly, we would need to search for that type everywhere it was referenced in the schema and make sure those queries, mutations, or other fields were restricted. For instance, if we only wanted logged-in users to be able to view user data, then we'd look for `User` in the above and below parts of the schema:

```
type Query {
  hello(date: Date!): String!
  isoString(date: Date!): String!
  reviews: [Review!]!
  me: User
  user(id: ID!): User
  searchUsers(term: String!): [UserResult!]!
}

type Mutation {
  createReview(review: CreateReviewInput!): Review
  createUser(user: CreateUserInput!, secretKey: String!): User
}
```

We would need to restrict `Review.author`, `Query.user`, and `Query.searchUsers`, and make sure that:

- `Query.me`, which returns a `User`, only returns the current user.
- `Mutation.createUser`, which also returns a `User`, doesn't return any user but the one just created by that client.

That's all for authorization on data access. The other part is authorization on running functions—specifically, functions that change things. While it's possible for a `Query` resolver function to change something, it's better to make those functions `Mutations`. Let's assume we've defined our `Query` and `Mutation` types properly, and haven't accidentally modified data in our `Query` resolvers. That means we only need to check our mutations, `createReview` and `createUser`. `createUser` we already [protected with a secretKey](#). `createReview` can currently be run by anyone, but we want it to be run only by logged-in users. Let's fix that:

`src/resolvers/Review.js`

```
import { ForbiddenError } from 'apollo-server'

export default {
  Query: ...
  Review: ...
}
```

```

Mutation: {
  createReview: (_, { review }, { dataSources, user }) => {
    if (!user) {
      throw new ForbiddenError('must be logged in')
    }

    dataSources.reviews.create(review)
  }
}
}

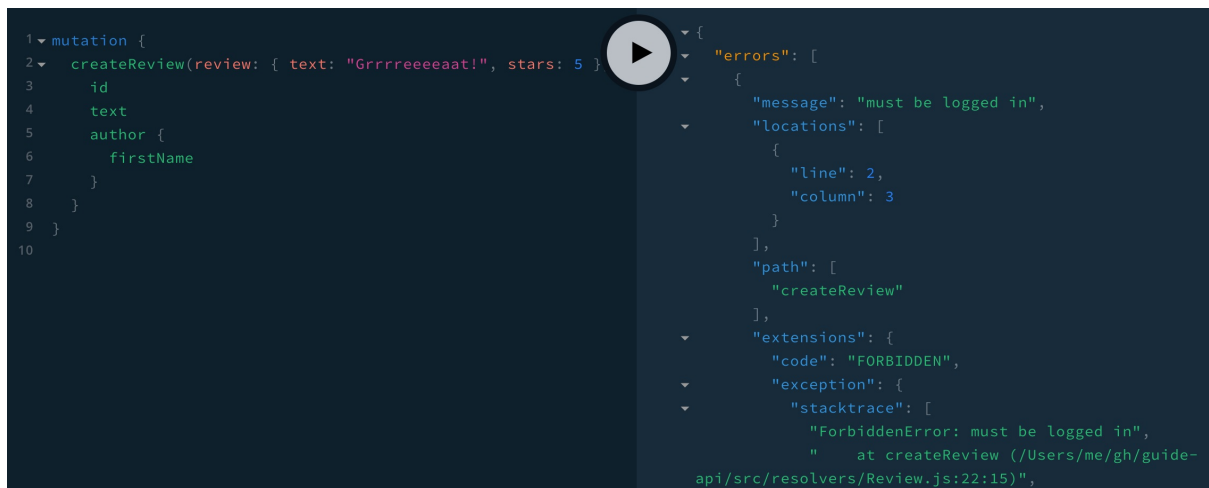
```

Now when we try the mutation without an authorization header, we get an error with the message `"must be logged in"` and code `"FORBIDDEN"`:

```

mutation {
  createReview(review: { text: "Grrrrreeeeaat!", stars: 5 }) {
    id
    text
    author {
      firstName
    }
  }
}

```



That concludes authorization in resolvers. Next let's check arguments. First we look back at the schema and think about which Query arguments need further validation:

```

type Query {
  hello(date: Date!): String!
  isoString(date: Date!): String!
  reviews: [Review!]!
  me: User
  user(id: ID!): User
  searchUsers(term: String!): [UserResult!]!
}

```

We don't need to do anything with the first two queries—our custom scalar checks validity, and any valid date is fine for those queries. The third and fourth don't have arguments. The last two do. Here's what they currently look like:

[src/resolvers/User.js](#)

```

export default {
  Query: {
    me: ...
    user: (_, { id }, { dataSources }) =>
      dataSources.users.findOneById(ObjectId(id)),
    searchUsers: (_, { term }, { dataSources }) =>

```

```

    dataSources.users.search(term)
  },
  ...
}

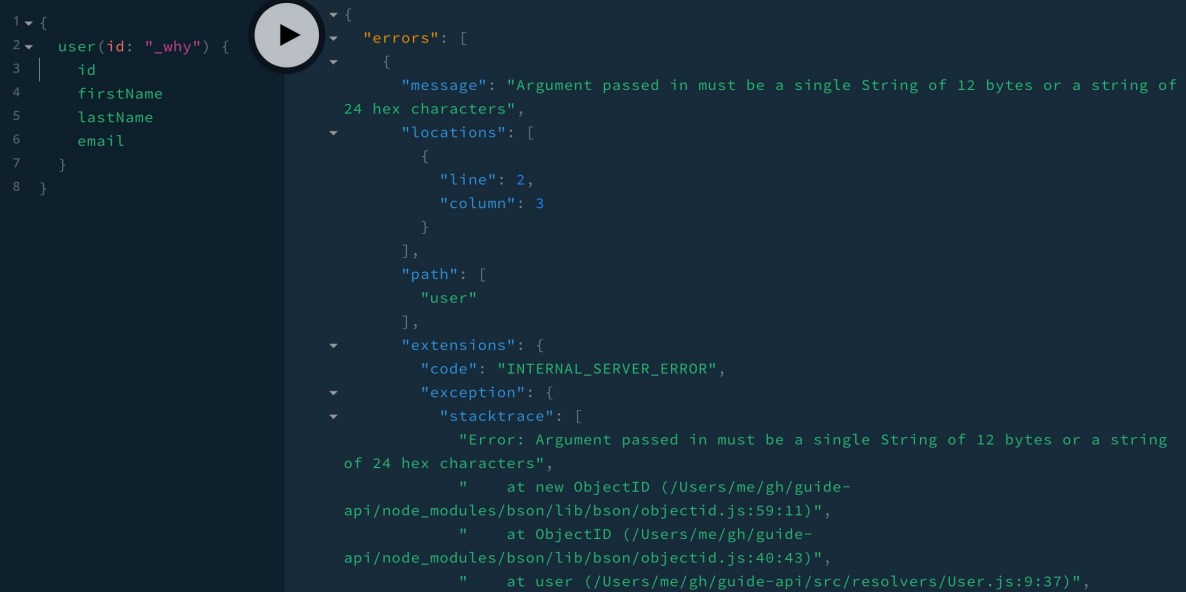
```

Searching with an empty string gives back an empty list, which is fine. We also don't need to worry about a NoSQL injection attack with a text search. So let's leave the searching to handle any string, blank or malicious, and move on to `query.user`. Any string validates as an `ID`, so let's see what happens when we try to get a user with an ID of `'_why'`:

```

{
  user(id: "_why") {
    firstName
  }
}

```



The screenshot shows a GraphQL IDE with a query on the left and its JSON response on the right. The query is:

```

1 {
2   user(id: "_why") {
3     id
4     firstName
5     lastName
6     email
7   }
8 }

```

The response is a JSON object with an `errors` array containing one error object:

```

{
  "errors": [
    {
      "message": "Argument passed in must be a single String of 12 bytes or a string of 24 hex characters",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "user"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "exception": {
          "stacktrace": [
            "Error: Argument passed in must be a single String of 12 bytes or a string of 24 hex characters",
            "    at new ObjectID (/Users/me/gh/guide-api/node_modules/bson/lib/bson/objectid.js:59:11)",
            "    at ObjectID (/Users/me/gh/guide-api/node_modules/bson/lib/bson/objectid.js:40:43)",
            "    at user (/Users/me/gh/guide-api/src/resolvers/User.js:9:37)",
          ]
        }
      }
    }
  ]
}

```

We get an error with the message `"Argument passed in must be a single String of 12 bytes or a string of 24 hex characters"` and code `"INTERNAL_SERVER_ERROR"`. We can tell from the stack trace that it's coming from our `objectId(id)` call, but it may very well be confusing to the client. Let's help the client out by giving them a better error message:

`src/resolvers/User.js`

```

import { UserInputError } from 'apollo-server'

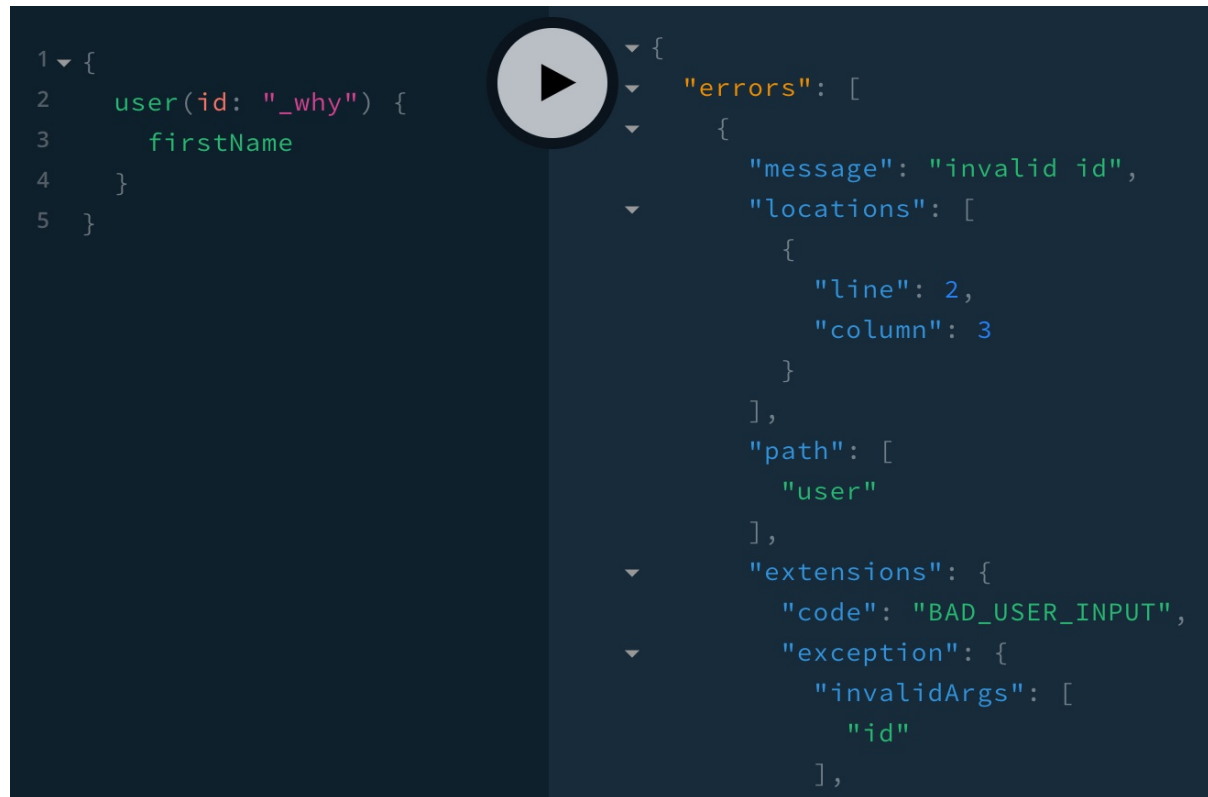
const OBJECT_ID_ERROR =
  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'

export default {
  Query: {
    me: ...
    user: (_, { id }, { dataSources }) => {
      try {
        return dataSources.users.findOneById(ObjectId(id))
      } catch (error) {
        if (error.message === OBJECT_ID_ERROR) {
          throw new UserInputError('invalid id', {
            invalidArgs: ['id']
          })
        } else {
          throw error
        }
      }
    }
  }
}

```

```
},
```

We use another built-in error type called `UserInputError`, which sets `extensions.code` to `BAD_USER_INPUT` and lists the invalid arguments in `extensions.invalidArgs`:



We're done checking Query arguments. Now let's do Mutation arguments:

```
type Mutation {
  createUser(user: CreateUserInput!, secretKey: String!): User
  createReview(review: CreateReviewInput!): Review
}
```

Because of `secretKey`, we can trust that our own code is the only one calling `createUser`. Let's also trust that our code sends good data for the `user` argument, so we can leave that resolver alone. Lastly is `createReview` with `CreateReviewInput`:

```
input CreateReviewInput {
  text: String!
  stars: Int
}
```

Inside our resolver, we can trust that `review.text` is a string and that `review.stars` is either undefined or an integer. We need to further check that `review.text` is a valid length (let's say at least two characters 🤔) and that `review.stars` is between 0 and 5.

`src/resolvers/Review.js`

```
import { ForbiddenError, UserInputError } from 'apollo-server'

const MIN_REVIEW_LENGTH = 2
const VALID_STARS = [0, 1, 2, 3, 4, 5]

export default {
  Query: ...
```

```

Review: ...
Mutation: {
  createReview: (_, { review }, { dataSources, user }) => {
    if (!user) {
      throw new ForbiddenError('must be logged in')
    }

    if (review.text.length < MIN_REVIEW_LENGTH) {
      throw new UserInputError(
        `text must be at least ${MIN_REVIEW_LENGTH} characters`,
        { invalidArgs: ['text'] }
      )
    }

    if (review.stars && !VALID_STARS.includes(review.stars)) {
      throw new UserInputError(`stars must be between 0 and 5`, {
        invalidArgs: ['stars']
      })
    }

    return dataSources.reviews.create(review)
  }
}
}

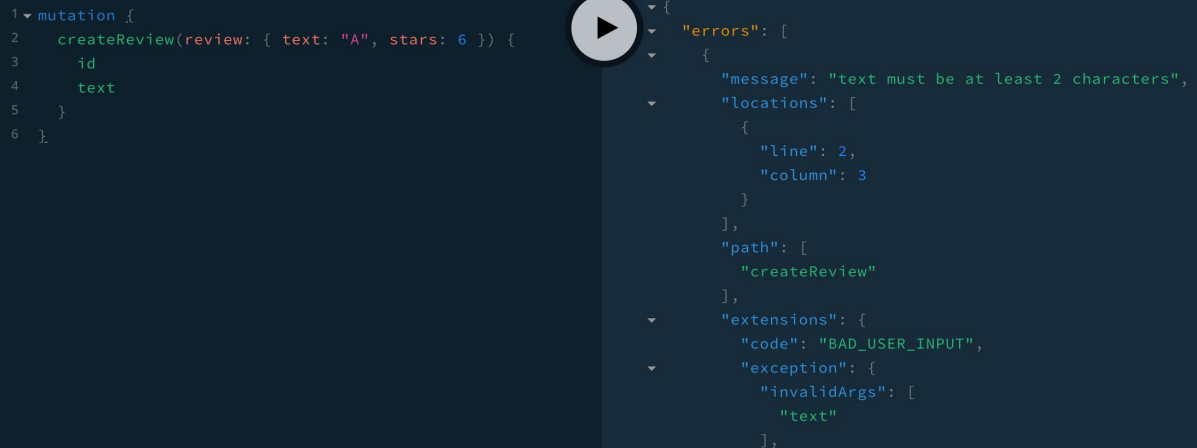
```

Since `CreateReviewInput!` is non-null, we don't have to check that `review` is defined. Similarly, we don't have to check that `review.text` is defined. Let's check both errors:

```

mutation {
  createReview(review: { text: "A", stars: 6 }) {
    id
    text
  }
}

```



```

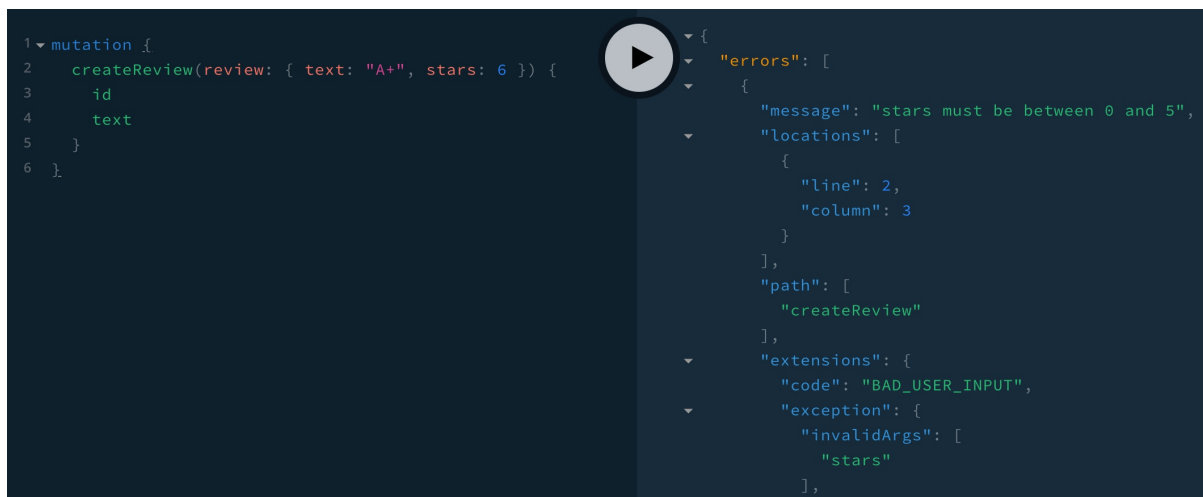
1 mutation {
2   createReview(review: { text: "A", stars: 6 }) {
3     id
4     text
5   }
6 }

```

```

{
  "errors": [
    {
      "message": "text must be at least 2 characters",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "createReview"
      ],
      "extensions": {
        "code": "BAD_USER_INPUT",
        "exception": {
          "invalidArgs": [
            "text"
          ]
        }
      }
    }
  ]
}

```



That's all of our input validation, and the last of our error checking! ✅

## Custom errors

If you're jumping in here, `git checkout 16_0.2.0` (tag `16_0.2.0`, or compare `16...17`)

In addition to the built-in `UserInputError`, `ForbiddenError`, and `AuthenticationError` that we've used, there's also their superclass, `ApolloError`, which we can use directly to add arbitrary error data or extend to make our own error classes. We'll do both in this section.

In the last section, when checking the `review` argument to `createReview`, we threw an error for either `review.text` or `review.stars`. If both were incorrect, the client would just get the first error, for `review.text`. Once the client fixed that and tried again, they would then get the `review.stars` error. It would be helpful to the client if we can give both errors at the same time.

We could do `{ invalidArgs: ['text', 'stars'] }` and combine the two error messages into one message, but it would be better to associate each error message with the corresponding argument—that way, for instance, the client can display individual error messages next to each invalid form field. It turns out that `UserInputError` takes any object as its second argument (and adds it to the response JSON's `extensions.exception`). Let's keep the recommended `invalidArgs` attribute, but change the value from an array to an object:

```

{
  invalidArgs: {
    text: 'must be at least 2 characters',
    stars: 'must be between 0 and 5'
  }
}

```

To get this, we update the code to:

`src/resolvers/Review.js`

```

import { isEmpty } from 'lodash'

export default {
  Query: ...
  Review: ...
  Mutation: {
    createReview: (_, { review }, { dataSources, user }) => {
      if (!user) {
        throw new ForbiddenError('must be logged in')
      }

      const errors = {}

```

```

    if (review.text.length < MIN_REVIEW_LENGTH) {
      errors.text = `must be at least ${MIN_REVIEW_LENGTH} characters`
    }

    if (review.stars && !VALID_STARS.includes(review.stars)) {
      errors.stars = `must be between 0 and 5`
    }

    if (!isEmpty(errors)) {
      throw new UserInputError('invalid review', { invalidArgs: errors })
    }

    return dataSources.reviews.create(review)
  }
}
}

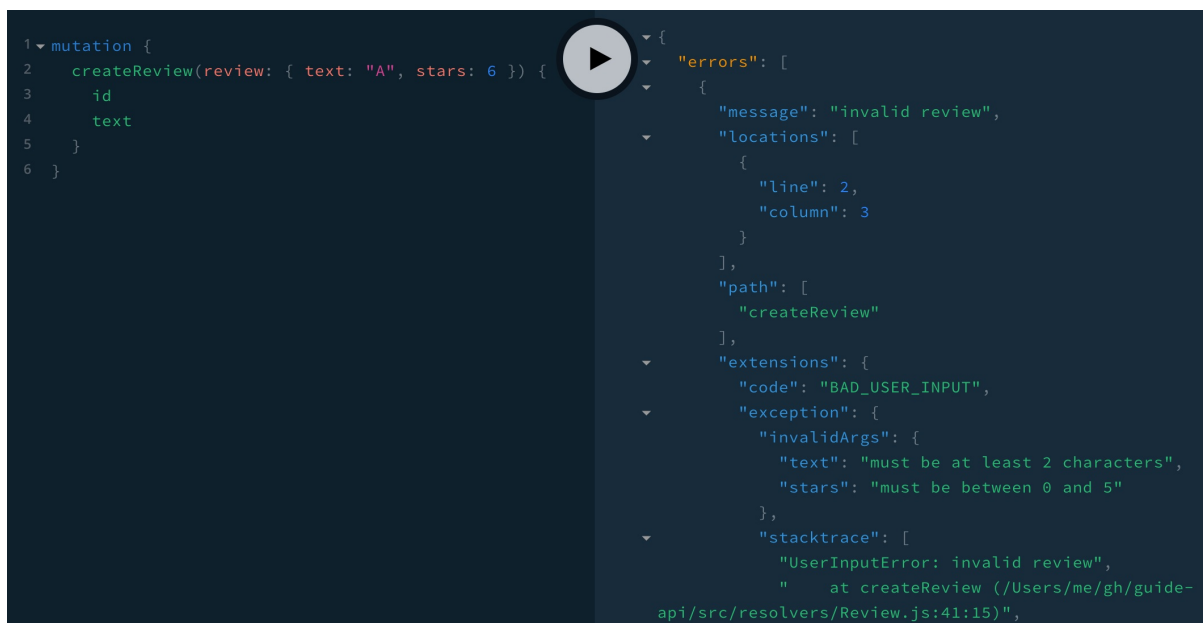
```

Now we see both errors together!

```

mutation {
  createReview(review: { text: "A", stars: 6 }) {
    id
    text
  }
}

```



We use `UserInputError` in one other place. Let's update the `invalidArgs` format there as well to be consistent so that the client can easily programmatically work with `extensions.exception.invalidArgs`:

`src/resolvers/User.js`

```

if (error.message === OBJECT_ID_ERROR) {
  throw new UserInputError('invalid id', {
    invalidArgs: { id: 'not a valid Mongo ObjectId' }
  })
}

```

We'll come back to `UserInputError` in a bit. For now let's consider this from `src/formatError.js`:

```

return new Error('Internal server error')

```

The resulting response is bare, without even a stack trace:



```

1 mutation {
2   createReview(review: { text: "A", stars: 6 }) {
3     id
4     text
5   }
6 }

```

```

{
  "errors": [
    {
      "message": "Internal server error"
    }
  ],
  "data": {
    "createReview": null
  }
}

```

Apollo only adds the `extensions` field (including a stack trace in development) for `ApolloError` and its subclasses. So let's use that:

`src/formatError.js`

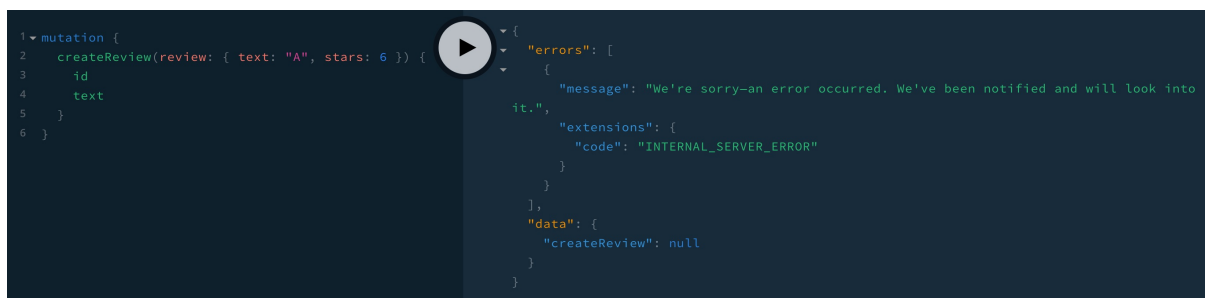
```

import { ApolloError } from 'apollo-server'

...

if (name.startsWith('Mongo')) {
  return new ApolloError(
    'We're sorry—an error occurred. We've been notified and will look into it.',
    'INTERNAL_SERVER_ERROR'
  )
} else {
  return error
}

```



```

1 mutation {
2   createReview(review: { text: "A", stars: 6 }) {
3     id
4     text
5   }
6 }

```

```

{
  "errors": [
    {
      "message": "We're sorry—an error occurred. We've been notified and will look into it.",
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR"
      }
    }
  ],
  "data": {
    "createReview": null
  }
}

```

`ApolloError` takes three arguments: the error message, a code, and additional properties to add to `extensions.exception`. We're using the first two. Having a code makes it easy for the client to handle all internal server errors similarly. Having a user-friendly message means that the client can show it directly to the user.

In case we want to throw an internal server error elsewhere in the future, let's make our own `InternalServerError` class:

`src/util/errors.js`

```

import { ApolloError } from 'apollo-server'

export class InternalServerError extends ApolloError {
  constructor() {
    super(
      'We're sorry—an error occurred. We've been notified and will look into it.',
      'INTERNAL_SERVER_ERROR'
    )

    Object.defineProperty(this, 'name', { value: 'InternalServerError' })
  }
}

```

`super()` gets the same arguments that the `ApolloError()` constructor got. The last thing is setting the object's name, which is used at the beginning of the stack trace. Now our use of the error can be simplified:



src/formatError.js

```
import { InternalServerError } from '../util/errors'

...

if (name.startsWith('Mongo')) {
  return new InternalServerError()
} else {
  return error
}
}
```

Let's also make a custom input error. Currently we're using `UserInputError` like this:

src/resolvers/Review.js

```
import { UserInputError } from 'apollo-server'

...

if (!isEmpty(errors)) {
  throw new UserInputError('invalid review', { invalidArgs: errors })
}
```

It would be simpler if we had an `InputError` class that we could use like this:

```
import { InputError } from '../util/errors'

...

if (!isEmpty(errors)) {
  throw new InputError({ review: errors })
}
```

And then `InputError` could take care of the error message for us. We could also use it in our user resolver:

src/resolvers/User.js

```
import { InputError } from '../util/errors'

export default {
  Query: {
    me: ...
    user: (_, { id }, { dataSources }) => {
      try {
        return dataSources.users.findOneById(ObjectId(id))
      } catch (error) {
        if (error.message === OBJECT_ID_ERROR) {
          throw new InputError({ id: 'not a valid Mongo ObjectId' })
        } else {
          throw error
        }
      }
    }
  },
}
```

The only difference here is that our argument `id` is a scalar type, so we pass `{ id: 'not a valid Mongo ObjectId' }` to `InputError()`, versus the `review` object type argument to `createReview`, which looked like:

```
{
  review: {
    text: 'must be at least 2 characters',
    stars: 'must be between 0 and 5'
  }
}
```

So when we implement our `InputError` class, we have to cover both scenarios—scalar arguments and their messages, as well as object arguments and their invalid field messages. As before, we subclass `ApolloError`, but this time the constructor creates the error `message`:

`src/util/errors.js`

```
export class InputError extends ApolloError {
  constructor(errors) {
    let messages = []

    for (const arg in errors) {
      if (typeof errors[arg] === 'string') {
        // scalar argument
        const errorReason = errors[arg]
        messages.push(`Argument ${arg} is invalid: ${errorReason}.`)
      } else {
        // object argument
        const errorObject = errors[arg]
        for (const prop in errorObject) {
          const errorReason = errorObject[prop]
          messages.push(`Argument ${arg}.${prop} is invalid: ${errorReason}.`)
        }
      }
    }

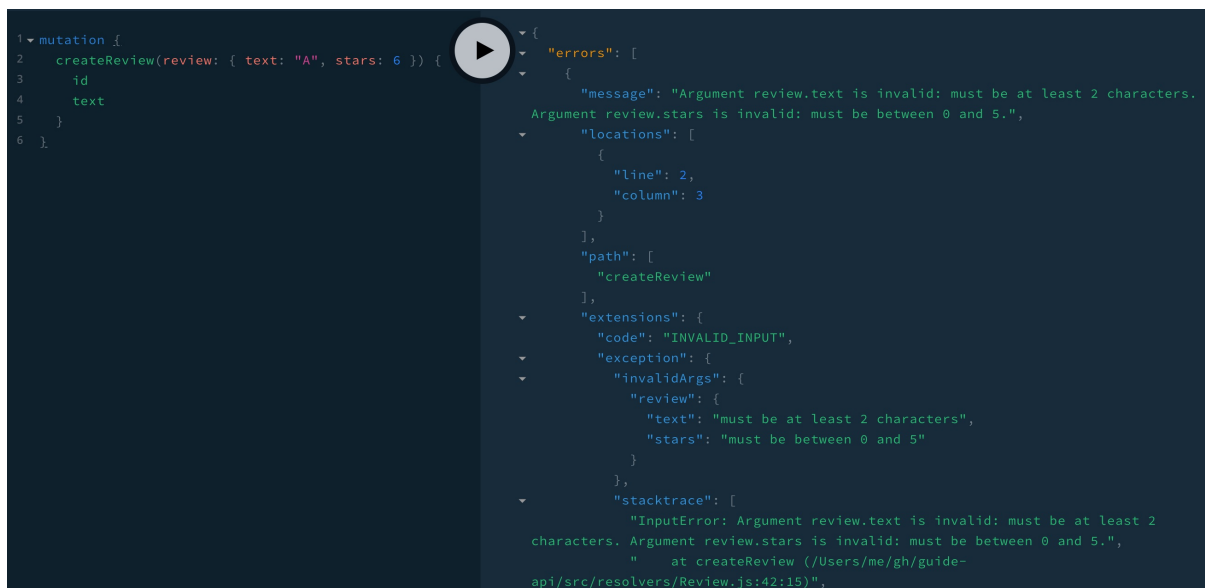
    const fullMessage = messages.join(' ')

    super(fullMessage, 'INVALID_INPUT', { invalidArgs: errors })

    Object.defineProperty(this, 'name', { value: 'InputError' })
  }
}
```

Now when we make an invalid query, we see:

- a very detailed error `message`
- our own error code `INVALID_INPUT`
- a different `invalidArgs` object, from which we can tell what argument the fields `text` and `stars` are on ( `review` )
- “InputError” at the beginning of the stack trace



In this section we went over:

- passing arbitrary `extensions.exception` properties as the second argument to `UserInputError()` (or the third

argument of `ApolloError()` )

- using `ApolloError()` directly
- creating our own error classes: `InternalServerError` and `InputError`

## Subscriptions

- [githubStars](#)
- [reviewCreated](#)

GraphQL subscriptions, along with the rest of the spec, are transport-agnostic: that is, the two parties communicating GraphQL don't need to use a specific way of sending messages. You can even do GraphQL with your friend by passing paper notes back and forth 😊.

The transport we've been using (HTTP) won't work for subscriptions because HTTP is unidirectional—only the client can initiate messages to the server, and the server only has a single opportunity to respond. We need a bidirectional transport—the client needs to be able to tell the server to start and stop the subscription, and the server needs to send subscription events. The main bidirectional transport used in web programming (and most often used for GraphQL subscriptions) is WebSockets.

In HTTP/2, the server can push resources to the client, but not messages to client code. With SSE ([Server-sent events](#)), the server can send messages to the client, and if we combine it with HTTP/2, we can do bidirectional communication over a single connection. However, WebSockets are more widely supported and easier to set up.

Subscriptions over WebSockets is supported by Apollo Server (at

`ws://hostname/graphql` — `ws://localhost:4000/graphql` in development). In the next section, we'll see what that looks like with a simple example. Then in [Review updates](#) we'll code a more complex example.

### githubStars

If you're jumping in here, `git checkout 17_0.2.0` (tag `17_0.2.0`, or compare `17...18`)

The simplest subscription used on the Guide site is for a single integer—the number of stars on the [GraphQLGuide/guide](#) repo. As always, we start with the schema:

`src/schema/Github.graphql`

```
type Subscription {
  githubStars: Int!
}
```

This means that each subscription event that the server sends the client will contain a single integer and be in this format:

```
{
  "data": {
    "githubStars": <integer>
  }
}
```

We include our new `.graphql` file by adding this to the bottom of `schema.graphql` :

`src/schema/schema.graphql`

```
#import 'Github.graphql'
```

We need a publish and subscribe system to keep track of which clients to send events to. Apollo Server has an interface that all pub/sub packages implement, so whichever we use, the API will be the same. We create an instance of the `PubSub` class, use its `.asyncIterator()` method in the subscription resolver, and its `.publish()` method to send events. Let's start with the first step, using the in-memory, for-use-in-development version of `PubSub` included in Apollo Server:

`src/util/pubsub.js`

```
import { PubSub } from 'apollo-server'

export const pubsub = new PubSub()
```

Our resolver is:

`src/resolvers/Github.js`

```
import { pubsub } from '../util/pubsub'

export default {
  Subscription: {
    githubStars: {
      subscribe: () => pubsub.asyncIterator('githubStars')
    }
  }
}
```

For subscriptions, instead of defining the function on `Subscription.field`, we use `Subscription.field.subscribe` and return an iterator. We're naming the iterator `'githubStars'`, so to send events to the iterator, we'll do `pubsub.publish('githubStars', { githubStars: 1337 })`.

Next we include the resolver:

`src/resolvers/index.js`

```
...

import Github from './Github'

export default [resolvers, Review, User, Date, Github]
```

Now where do we call `pubsub.publish()`? We have to get the information first. Where do we get it from? GitHub, of course! The first three versions of their API were REST-based, but their v4 is a GraphQL API—let's use that. [Their docs](#) say the endpoint is `https://api.github.com/graphql` and that we need to [create an access token](#) to use the API. Once we've done that, we add a new `GITHUB_TOKEN` environment variable with the token we created:

`.env`

```
SECRET_KEY=9e769699fae6f594beafb46e9078c2
GITHUB_TOKEN=...
```

Now we can use `process.env.GITHUB_TOKEN` in our auth header to the GitHub API. Let's put our code in the `data-sources/` directory. Even though it doesn't talk to our database or follow Apollo's `DataSource` API (since we don't need context, a new instance for every request, batching, or caching), it is a source of data used in our app.

`src/data-sources/Github.js`

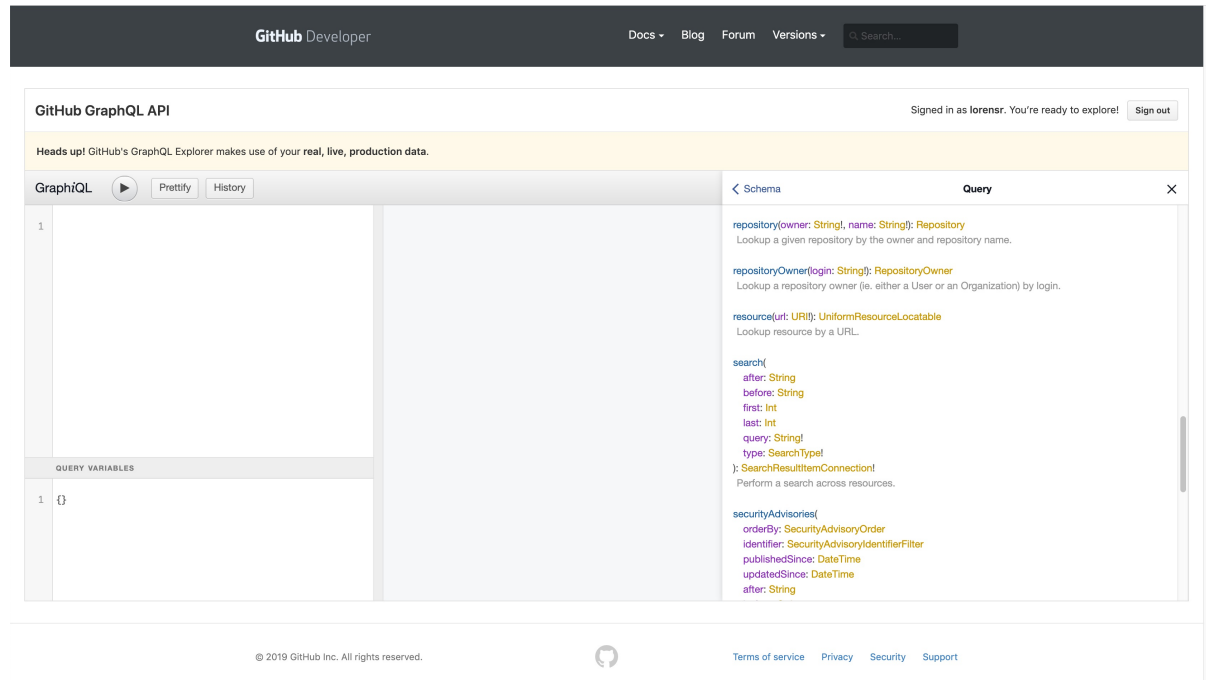
```
import { GraphQLClient } from 'graphql-request'

const githubAPI = new GraphQLClient('https://api.github.com/graphql', {
  headers: {
    authorization: `bearer ${process.env.GITHUB_TOKEN}`
  }
})
```

```
}
})
```

The simplest way to make GraphQL requests is with the `graphql-request` npm package. Now we can call `githubAPI.request(queryString)`, and our query will be sent to GitHub with our auth header.

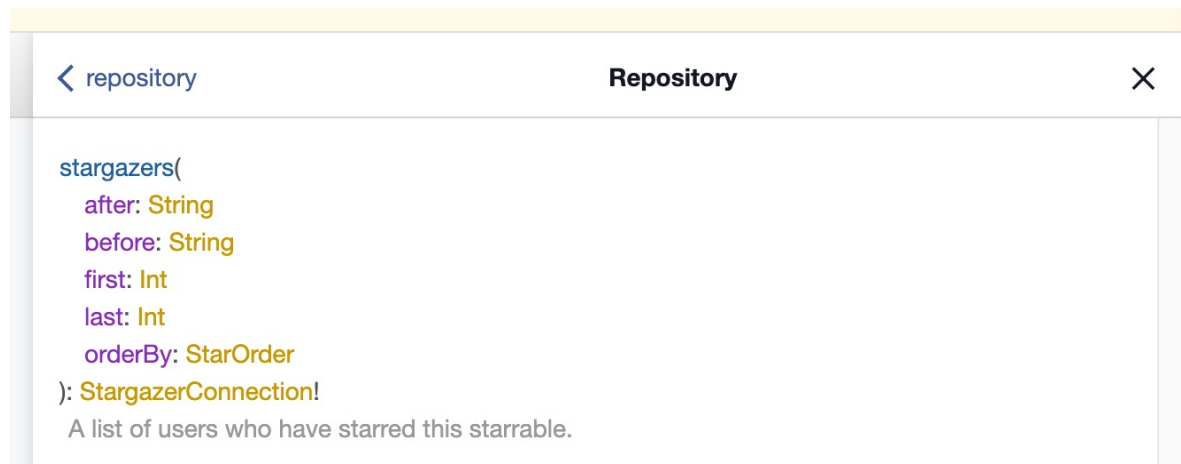
To determine what our query should be, we can browse GitHub's GraphQL Explorer (an authenticated GraphiQL). A repo's star count should be included in a repository's information, so let's look for a root Query field for getting a repository:



We find:

```
# Lookup a given repository by the owner and repository name.
repository(owner: String!, name: String!): Repository
```

Clicking on the `Repository` type gives us a long list of fields, including a `stargazers` field:



And clicking on the `StargazerConnection` type gives us:

[< Repository](#)**StargazerConnection****🔍 Search StargazerConnection...**

The connection type for User.

**FIELDS**

**edges:** [\[StargazerEdge\]](#)

A list of edges.

**nodes:** [\[User\]](#)

A list of nodes.

**pageInfo:** [PageInfo!](#)

Information to aid in pagination.

**totalCount:** [Int!](#)

Identifies the total count of items in the connection.

And we find that `totalCount` is the field we need. Putting all of that together gives us:

```
const GUIDE_STARS_QUERY = `
query GuideStars {
  repository(owner: "GraphQLGuide", name: "guide") {
    stargazers {
      totalCount
    }
  }
}
```

We can make this query periodically to keep the count up to date. Let's create a `startPolling()` function that does that. When it gets a new number, it will call `pubsub.publish()` :

`src/data-sources/Github.js`

```
import { pubsub } from '../util/pubsub'

...

export default {
```

```

async fetchStarCount() {
  const data = await githubAPI.request(GUIDE_STARS_QUERY).catch(console.log)
  return data && data.repository.stargazers.totalCount
},

startPolling() {
  let lastStarCount

  setInterval(async () => {
    const starCount = await this.fetchStarCount()
    const countChanged = starCount && starCount !== lastStarCount

    if (countChanged) {
      pubsub.publish('githubStars', { githubStars: starCount })
      lastStarCount = starCount
    }
  }, 1000)
}
}

```

The first argument to `pubsub.publish()` is the name of the async iterator and the second argument is the event data, the format of which needs to match our Subscription field in the schema ( `type Subscription { githubStars: Int! }` ).

Next we need to call `startPolling()` on startup. The place where all the other data sources are included seems a fitting place:

`src/data-sources/index.js`

```

import Github from './Github'

Github.startPolling()

```

The last change we need to make is to our context function:

`src/context.js`

```

export default async ({ req }) => {
  const context = {}

  const jwt = req.headers.authorization

```

We're getting a `req` argument and assuming that it has `headers.authorization` properties. But actually, `req` will be undefined for subscriptions. So let's guard against that:

```

export default async ({ req }) => {
  const context = {}

  const jwt = req && req.headers.authorization

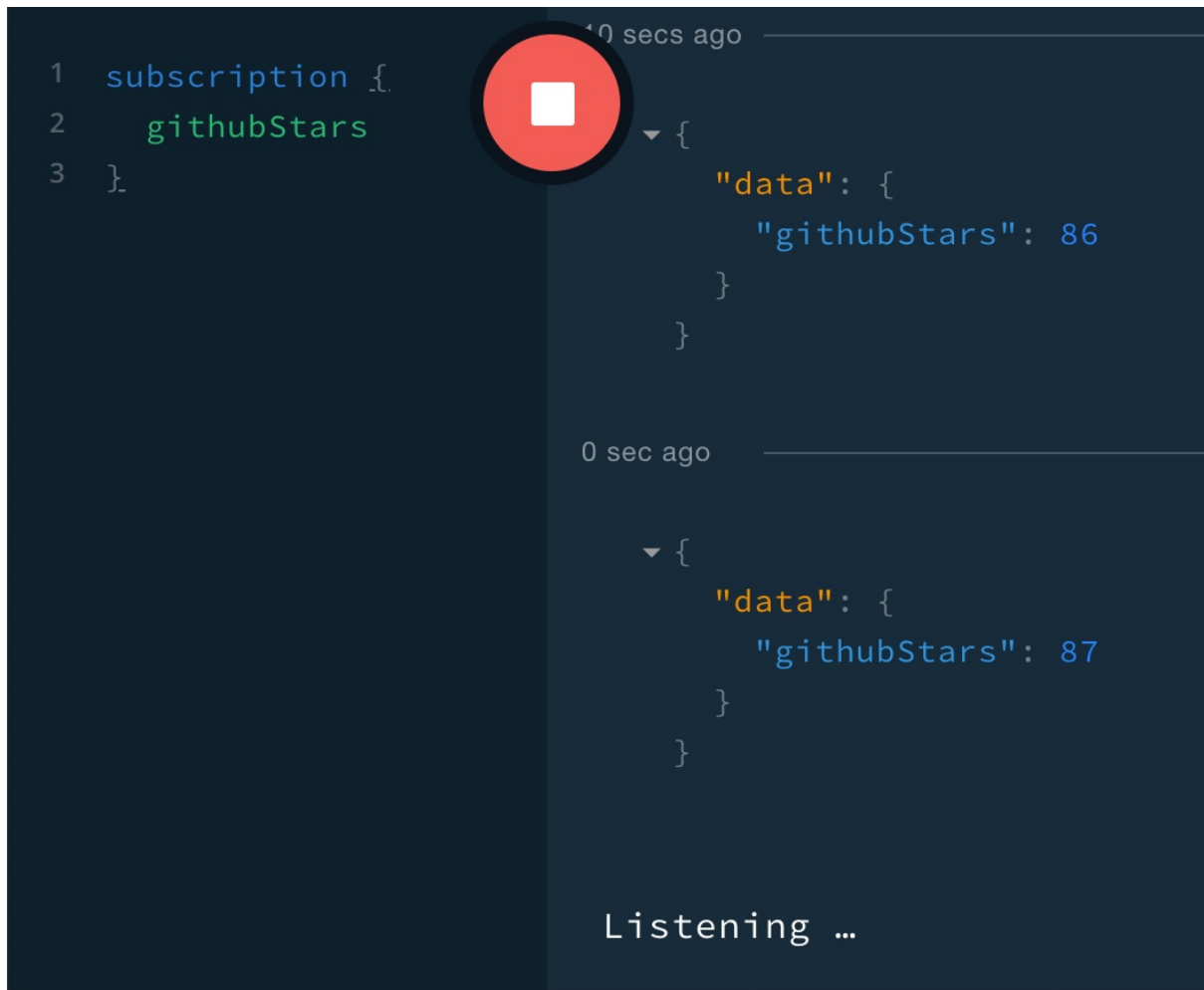
```

Now we test out our new subscription:

```

subscription {
  githubStars
}

```



When we hit the play button, it turns red, but nothing appears on the right—that's because we haven't received an event from the server yet, because the server only publishes when the value changes. But if we star [the repo](#), we'll see an event of the form:

```
{
  "data": {
    "githubStars": 87
  }
}
```

And when we unstar the repo, we see another event with the number one lower. Great, we've got realtime updates!



Well... depending on your definition of realtime. Since we're polling once a second, we might lag around a second. In the next section we'll see even faster updates, where the publish happens as soon as the server receives a user's action.

Lastly, let's see what the WebSocket communication looks like. If we open devtools Network tab, hit the stop button in Playground, hit play, unstar and re-star the repo, select the `graphql` item in the list on the bottom-left, and select the Messages tab, we'll see something like:



Name	× Headers	Messages	Cookies	Timing
graphql	All	Enter regex, for example: (web)?socket		
		Data	Length	Time
localhost		↑{"type":"connection_init","payload":{}}	39	00:06:53.679
localhost		↑{"id":"1","type":"start","payload":{"variables":{"githubStars":86}},"extensions":{"githubStars":87}}}	134	00:06:53.679
localhost		↓{"type":"connection_ack"}	25	00:06:53.680
localhost		↓{"type":"data","id":"1","payload":{"data":{"githubStars":86}}}	62	00:06:59.575
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":87}}}	62	00:07:00.581
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":88}}}	62	12:03:42.581
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":89}}}	62	12:51:22.842
▼ {id: "1", type: "start", ...} id: "1" ▼ payload: {variables: {}, extensions: {}, operationName: null, query: "subscription { extensions: {} operationName: null query: "subscription { githubStars }" variables: {} type: "start" }				

The rows with the green up arrow are messages sent over the WebSocket to the server, and the rows with the red down arrow are messages sent from the server to the browser. When we hit the play button, Playground opens the connection to `ws://localhost:4000/graphql` and sends two messages: one with type `connection_init` and one with:

- `type: "start"` —We're starting a subscription.
- `payload.query` —The GraphQL document containing our subscription (what we typed on the left side of the Playground).
- `id: 1` —We might start more subscriptions over this websocket, so we have a number to identify this one that we're starting in this message.

Then the server sends a message with type `connection_ack` (acknowledging receipt of the `connection_init`), and messages like this:

Name	× Headers	Messages	Cookies	Timing
graphql	All	Enter regex, for example: (web)?socket		
		Data	Length	Time
localhost		↑{"type":"connection_init","payload":{}}	39	00:06:53.679
localhost		↑{"id":"1","type":"start","payload":{"variables":{"githubStars":86}},"extensions":{"githubStars":87}}}	134	00:06:53.679
localhost		↓{"type":"connection_ack"}	25	00:06:53.680
localhost		↓{"type":"data","id":"1","payload":{"data":{"githubStars":86}}}	62	00:06:59.575
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":87}}}	62	00:07:00.581
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":88}}}	62	12:03:42.581
		↓{"type":"data","id":"1","payload":{"data":{"githubStars":89}}}	62	12:51:22.842
▼ {type: "data", id: "1", payload: {data: {githubStars: 89}}} id: "1" ▶ payload: {data: {githubStars: 89}} type: "data"				

- `type: "data"` —This message contains a subscription event.
- `id: 1` —This event corresponds to the subscription with an `id` of 1.
- `payload: {data: {githubStars: 89}}` —This is the subscription event, which Playground displays in the right-side panel.

Similar to how Playground took our subscription document and put it in WebSocket messages in the right format, and how it parsed the response messages and displayed the payload on the page, most of our clients will be using libraries that take care of the messaging part, so that all they'll get is the payload object: `{data: {githubStars: 89}}`.

## reviewCreated

If you're jumping in here, `git checkout 18_0.2.0` (tag `18_0.2.0`, or compare `18...19`)

In the last section we set up our first subscription for a single integer based on an external source of data. In this section we'll set up a subscription for an object type ( `Review` ) based on a user action (creating a review). The subscription will be named `reviewCreated`, and whenever any user creates a review, the server will send an event with that review data to all the clients that are subscribed to the `reviewCreated` subscription.

Let's start with the schema!

`src/schema/Review.graphql`

```
type Subscription {
  reviewCreated: Review!
}
```

We now have an error because we're declaring `type Subscription` in two places, so let's change the one in `Github.graphql` (which we can see in `src/schema/schema.graphql` is included after `Review.graphql` is included) to `extend type Subscription`:

`src/schema/Github.graphql`

```
extend type Subscription {
  githubStars: Int!
}
```

Now we only need to do two things:

- add a `Subscription.reviewCreated.subscribe` function that returns an iterator
- at the end of the `createReview` resolver, publish the new review object to that iterator

`src/resolvers/Review.js`

```
import { pubsub } from '../util/pubsub'

export default {
  Query: ...
  Review: ...
  Mutation: {
    createReview: (_, { review }, { dataSources, user }) => {
      ...

      const newReview = dataSources.reviews.create(review)

      pubsub.publish('reviewCreated', {
        reviewCreated: newReview
      })

      return newReview
    }
  },
  Subscription: {
    reviewCreated: { subscribe: () => pubsub.asyncIterator('reviewCreated') }
  }
}
```

The second argument to `pubsub.publish` is the event data, which needs to match the schema ( `reviewCreated: Review!` ): a `reviewCreated` attribute with an object of type `Review` for the value.

Aaaaand we're done! That was easy. To test, we start the subscription in one Playground tab:

```
subscription {
  reviewCreated {
```

```

    id
    text
    stars
    createdAt
  }
}

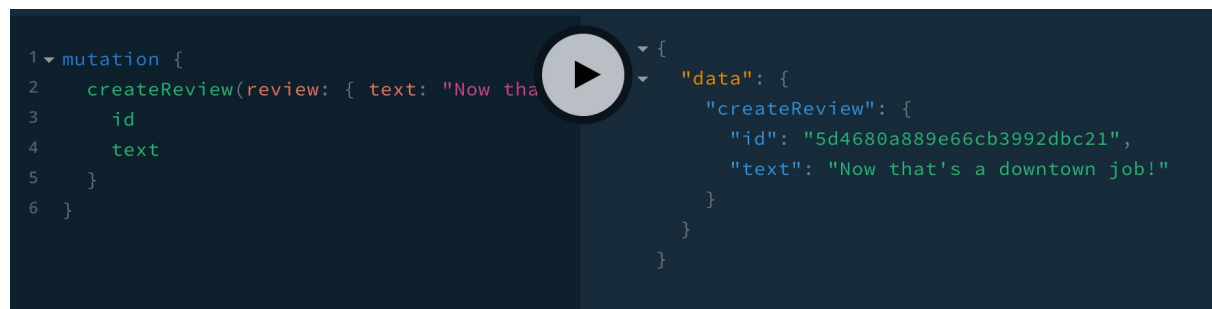
```

And create the review in another:

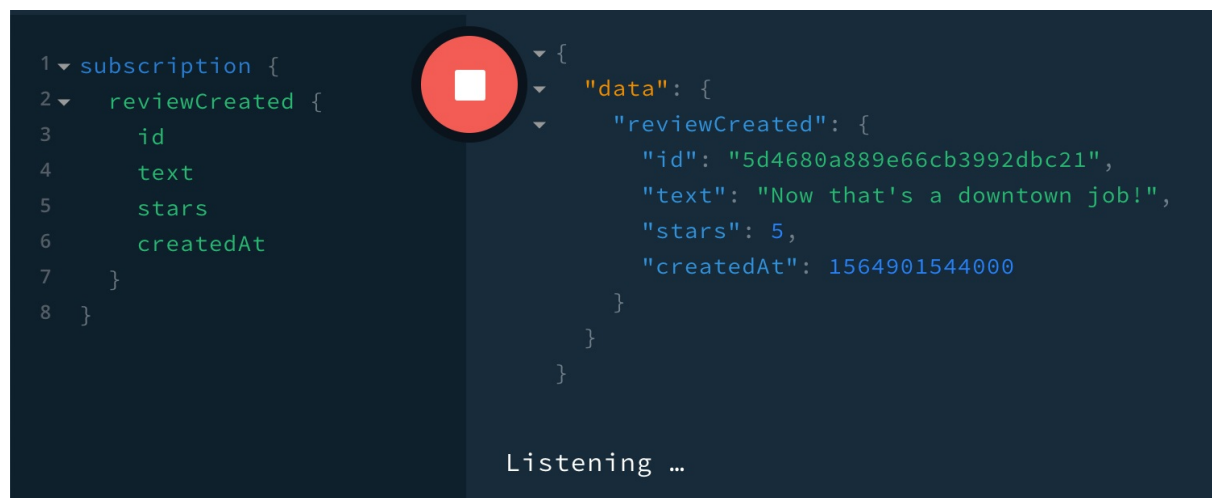
```

mutation {
  createReview(review: { text: "Now that's a downtown job!", stars: 5 }) {
    id
    text
  }
}

```



Now when we go back to the subscription tab, we'll see the event:



Other common types of subscriptions include when objects are edited and deleted:

```

type Subscription {
  reviewEdited: Review!
  reviewDeleted: ID!
}

```

`reviewEdited` events would include the review post-edit, and `reviewDeleted` events would just include the ID of the deleted review, so that clients can remove it from their cache. We'll discuss subscriptions in more depth in the section [Extended topics -> Subscription design](#).

## Testing

Background: [Testing](#)

- [Static testing](#)
- [Review integration tests](#)
- [Code coverage](#)
- [User integration tests](#)
- [Unit tests](#)
- [End-to-end tests](#)

In the Background chapter we go over [mocking](#) and [which types of tests](#) are best to write.

In this section we'll start out by setting up static testing. Next we'll write integration tests for our review operations. Then we'll check how much of the code we've tested using a code coverage tool. Next we'll fill in some of the coverage gaps with more integration tests and unit tests. Finally, we'll write an end-to-end test.

## Static testing

If you're jumping in here, `git checkout 19_0.2.0` (tag [19\\_0.2.0](#), or compare [19...20](#))

Static testing is done through linting, a type of static code analysis. It's called *static* because, unlike the tests we write code for, no code is being run during static testing—instead, a tool analyzes the code for certain types of mistakes that can be found by just looking at the code and not running it. One such mistake is when we use a variable without declaring it. In JavaScript, the main tool for static analysis is [ESLint](#), and here's a [list of possible rules](#)—things it can analyze that we can choose to disallow in our code.

We have `eslint` and `eslint-plugin-node` installed as dev dependencies, so all we need to do is configure ESLint:

`.eslintrc.js`

```
module.exports = {
  env: {
    es6: true,
    node: true
  },
  extends: 'plugin:node/recommended',
  parserOptions: {
    sourceType: 'module'
  }
}
```

`env` says that we're using ES6 in Node.js, `extends` says to use `eslint-plugin-node`'s set of recommended linting rules, and `sourceType: 'module'` means that we're using modules. We can add an npm script for linting:

`package.json`

```
{
  ...
  "scripts": {
    ...
    "lint": "eslint src/"
  }
}
```

When we try it out ( `npm run lint` ), we get errors saying:

```
error  Import and export declarations are not supported yet  node/no-unsupported-features/es-syntax
```

It's warning us that using the keywords `import` and `export` in our code won't work because it's not supported by node. Our code actually does work, because we're using babel. So let's disable this rule (the name of the rule is printed at the end):

`.eslintrc.js`

```
module.exports = {
  ...
  rules: {
    'node/no-unsupported-features/es-syntax': 0
  }
}
```

Now when we do `npm run lint`, it succeeds—no errors are found.

A common practice is setting up linting to occur as a *pre-commit hook*—that is, a command that will automatically be run whenever we enter `git commit`, and if the command fails, the commit will be canceled. The easiest way to set this up is with `husky`, one of our dev dependencies, which simply uses a `package.json` attribute:

`package.json`

```
{
  ...
  "husky": {
    "hooks": {
      "pre-commit": "npm run lint"
    }
  }
}
```

Now if we commit, we see that `eslint src/` is run before the commit happens:

```
$ git commit -m 'Set up linting'
husky > pre-commit (node v8.11.3)

> guide-api@0.1.0 lint /guide-api
> eslint src/

[20 bfe4bf1] Set up linting
2 files changed, 21 insertions(+), 1 deletion(-)
create mode 100644 .eslintrc.js
```

## Review integration tests

If you're jumping in here, `git checkout 20_0.2.0` (tag `20_0.2.0`, or compare `20...21`)

The different types of testing are basically defined by how much is mocked 🤖. In integration tests, we usually just mock network requests. The main type of network request our server makes is to the database, so we'll be mocking our MongoDB collection methods. We also won't need our tests to make network requests to the GraphQL server because Apollo has `createTestClient()` which allows us to query the server without starting the server. It puts our queries through the Apollo Server request pipeline as if they were HTTP requests.

`createTestClient()` returns an object with `query` and `mutate` functions, which each take a `GraphQLRequest` object:

`apollo-server-types`

```
export interface GraphQLRequest {
  query?: string;
  operationName?: string;
  variables?: VariableValues;
  extensions?: Record<string, any>;
  http?: Pick<Request, 'url' | 'method' | 'headers'>;
}
```

```
}

```

Usually we just use the `query` and `variables` properties, but we can also use `http`, for instance to include an authorization header:

```
const { query } = createTestClient(server)
query({
  query: gql`...`,
  http: {
    headers: {
      authorization: `Bearer ${token}`
    }
  }
})

```

Then the server would run our context function, decode the auth token, and add the user doc to the context that it gives to resolvers.

`createTestClient()` 's only parameter is an instance of Apollo Server, so our tests will need one. We can't use the one created in `src/index.js` because our tests will need to be able to modify data sources and context. So let's make a `createTestServer()` function. And let's create a new file that exports all of our testing helper functions and data, so that the test files can import whatever they need from one place:

`test/guide-test-utils.js`

```
import { ApolloServer } from 'apollo-server'

import { Reviews, Users } from '../src/data-sources/'
import {
  typeDefs,
  resolvers,
  context as defaultContext,
  formatError
} from '../src/'

export const createTestServer = ({ context = defaultContext } = {}) => {
  const reviews = new Reviews({})

  const users = new Users({})

  const server = new ApolloServer({
    typeDefs,
    resolvers,
    dataSources: () => ({ reviews, users }),
    context,
    formatError
  })

  return { server, dataSources: { reviews, users } }
}

export { createTestClient } from 'apollo-server-testing'
export { default as gql } from 'graphql-tag'

```

`createTestServer()` returns both the server instance and the data sources (so that tests can spy on or modify data source functions). In order for the above code to work, we need to add some exports:

`src/data-sources/index.js`

```
...

export { Reviews, Users, Github }

```

`src/index.js`

```
...
export { typeDefs, resolvers, context, formatError }
```

Now that we've got our `guide-test-utils.js` file, we can import from it into our test files. It would be nice if we could import without thinking about relative paths, as if it were a node module:

```
import {
  createTestServer,
  createTestClient,
  gql
} from 'guide-test-utils'
```

To enable this, we can create a config file:

`jest.config.js`

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

module.exports = {
  moduleDirectories: ['node_modules', path.join(__dirname, 'test')]
}
```

Jest will now look for modules both in `node_modules/` and in `test/`. (Jest, made by Facebook, is the most popular JavaScript testing framework.)

While it will run, it won't pass linting, which we'll find out either in our editor—if ESLint is enabled—or when we try to commit and it fails:

```
husky > pre-commit (node v8.11.3)

> guide-api@0.1.0 lint /guide-api
> eslint src/


/guide-api/src/resolvers/Review.test.js
  6:8  error  "guide-test-utils" is not found  node/no-missing-import

* 1 problem (1 error, 0 warnings)
```

ESLint is looking in our `node_modules/` to make sure that anything we import is there. But there is no `node_modules/guide-test-utils/`, so it gives an error. If we look at the documentation for the [node/no-missing-import rule](#), we learn that there's a way to tell it to look in additional locations for modules—in this case, we want it to look in the `./test` directory:

`.eslintrc.js`

```
module.exports = {
  ...
  rules: {
    ...
    'node/no-missing-import': [
      'error',
      {
        resolvePaths: ['./test']
      }
    ]
  }
}
```

Now committing or doing `npm run lint` succeeds .

Let's move on to writing the review tests themselves. Since the entry point to review operations and most of the logic is in the resolvers, let's put our test file next to the `Review.js` resolvers file, adding `.test` to the filename:

`src/resolvers/Review.test.js`

```
import {
  createTestServer,
  createTestClient,
  gql
} from 'guide-test-utils'

test('something', () => {
  const result = ...

  expect(result).toSomething()
})
```

Jest provides a set of [global functions](#), including the basic test function `test()` (or `it()`), in which we run part of our code and assert something about the result. We use `expect()` for assertions, which is followed by any of [a number of matcher methods](#), such as:

```
expect(result).toBeTruthy()
expect(result).toBe('this string')
expect(array).not.toContain(10)
expect(doSomething).toThrow('must be logged in')
```

We'll write two tests, one for each review operation (`reviews` query and `createReview` mutation):

```
import {
  createTestServer,
  createTestClient,
  gql
} from 'guide-test-utils'

test('reviews', () => {
})

test('createReview', () => {
})
```

For the first, we'll start by first creating a test server and then a test client:

```
import {
  createTestServer,
  createTestClient,
  gql
} from 'guide-test-utils'

test('reviews', async () => {
  const { server } = createTestServer()
  const { query } = createTestClient(server)

  const result = await query({ query: ... })
})
```

We need a query document to give to `query()`. To try to cover as many resolvers as possible, let's select all `Review` and `user` fields except `User.email` (it requires authentication, which we'll do in the second test).

```
const REVIEWS = gql`
  query {
    reviews {
      id
```



```

      text
      stars
      author {
        id
        firstName
        lastName
        username
        photo
        createdAt
        updatedAt
      }
      createdAt
      updatedAt
    }
  }
}

test('reviews', async () => {
  const { server } = createTestServer()
  const { query } = createTestClient(server)

  const result = await query({ query: REVIEWS })
})

```

This test will send the `REVIEWS` query via the test client to our server. But before we make an assertion and run our code, we have to mock the database! Specifically, we have to mock the collection functions that will be called when our query is run. Looking at `src/resolvers/Review.js`, we see that `dataSources.reviews.all` and `dataSources.users.findOneById` are called. They both call `this.collection.find().toArray()`, so we need to mock `.find().toArray()` for both collections, as well as `this.collection.createIndex()`, which we call in the `Users` data source constructor.

[test/guide-test-utils.js](#)

```

export const createTestServer = ({ context = defaultContext } = {}) => {
  const reviews = new Reviews({
    find: () => ({
      toArray: jest.fn().mockResolvedValue(mockReviews)
    })
  })

  const users = new Users({
    createIndex: jest.fn(),
    find: () => ({
      toArray: jest.fn().mockResolvedValue(mockUsers)
    })
  })

  const server = new ApolloServer({
    dataSources: () => ({ reviews, users }),
    ...
  })

  ...
}

```

We'll create a mock function using `jest.fn()`. By default it returns `undefined`, which works for `createIndex()`, but for `find()` we need to return an object that has a `toArray()` method that returns a Promise that resolves to an array of documents 🙄😬. We'll also need to create the `mockReviews` and `mockUsers` constants:

```

import { ObjectId } from 'mongodb'

const updatedAt = new Date('2020-01-01')

export const mockUser = {
  _id: ObjectId('5d24f846d2f8635086e55ed3'),
  firstName: 'First',
  lastName: 'Last',

```

```

    username: 'mockA',
    authId: 'mockA|1',
    email: 'mockA@gmail.com',
    updatedAt
  }

  const mockUsers = [mockUser]

  const reviewA = {
    _id: ObjectId('5ce6e47b5f97fe69e0d63479'),
    text: 'A+',
    stars: 5,
    updatedAt,
    authorId: mockUser._id
  }

  const reviewB = {
    _id: ObjectId('5cf8add4c872001f31880a97'),
    text: 'Passable',
    stars: 3,
    updatedAt,
    authorId: mockUser._id
  }

  const mockReviews = [reviewA, reviewB]

```

Now our `'reviews'` test should return `reviewA` and `reviewB`, both with author `mockUser`. Let's complete the test with an assertion:

```

test('reviews', async () => {
  const { server } = createTestServer()
  const { query } = createTestClient(server)

  const result = await query({ query: REVIEWS })
  expect(result).toMatchSnapshot()
})

```

To run the test, let's add an npm script:

`package.json`

```

{
  ...
  "scripts": {
    ...
    "test": "jest"
  }
}

```

Now when we do `npm run test` (or just `npm test`), Jest will find all `*.test.js` files and run the tests it finds inside them.

Our assertion `expect(result).toMatchSnapshot()` will save a snapshot (a serialization of the result, saved to a new `__snapshots__` directory). Whenever we get a different result from the saved snapshot, the test will fail, and we'll either need to fix the code or (in the case when the result is correctly different) tell Jest to update the snapshot.

Snapshots should be added to git.

```

$ npm test

> guide-api@0.1.0 test /guide-api
> jest

PASS src/resolvers/Review.test.js
  ✓ reviews (58ms)

> 1 snapshot written.

```

```
console.log src/index.js:22
GraphQL server running at http://localhost:4000/
```

#### Snapshot Summary

```
> 1 snapshot written from 1 test suite.
```

```
Test Suites: 1 passed, 1 total
```

```
Tests:      1 passed, 1 total
```

```
Snapshots:  1 written, 1 total
```

```
Time:       3.375s, estimated 4s
```

```
Ran all test suites.
```

```
Jest did not exit one second after the test run has completed.
```

This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jest with `--detectOpenHandles` to troubleshoot this issue.

To terminate the command, type `ctrl-c`.

We see that our one test passes, and a new snapshot is written. We can look at the file to make sure it's correct:

`src/resolvers/__snapshots__/Review.test.js.snap`

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`reviews 1`] = `
Object {
  "data": Object {
    "reviews": Array [
      Object {
        "author": Object {
          "createdAt": 1562703942000,
          "firstName": "First",
          "id": "5d24f846d2f8635086e55ed3",
          "lastName": "Last",
          "photo": "https://avatars.githubusercontent.com/u/1",
          "updatedAt": 1577836800000,
          "username": "mockA",
        },
        "createdAt": 1558635643000,
        "id": "5ce6e47b5f97fe69e0d63479",
        "stars": 5,
        "text": "A+",
        "updatedAt": 1577836800000,
      },
      Object {
        "author": Object {
          "createdAt": 1562703942000,
          "firstName": "First",
          "id": "5d24f846d2f8635086e55ed3",
          "lastName": "Last",
          "photo": "https://avatars.githubusercontent.com/u/1",
          "updatedAt": 1577836800000,
          "username": "mockA",
        },
        "createdAt": 1559801300000,
        "id": "5cf8add4c872001f31880a97",
        "stars": 3,
        "text": "Passable",
        "updatedAt": 1577836800000,
      },
    ],
  },
  "errors": undefined,
  "extensions": undefined,
  "http": Object {
    "headers": Headers {
      Symbol(map): Object {},
    },
  },
}
`;
```

That looks good! We've got what we expected in the `"data"` result attribute and nothing in the `"errors"` attribute. However, if we look at the end of the test output, we see a problem:

```
Jest did not exit one second after the test run has completed.
```

This usually means that there are asynchronous operations that weren't stopped in your tests. Consider running Jest with `--detectOpenHandles` to troubleshoot this issue.

It's saying we've started code running that hasn't stopped running. If we look above that, we see this output:

```
console.log src/index.js:22
  GraphQL server running at http://localhost:4000/
```

It looks like our non-test server is running—that's the running code that Jest is warning us about. So we need to edit `src/index.js` to not start the server during tests. Jest sets `NODE_ENV` to `'test'`, so let's use that:

`src/index.js`

```
const start = () => {
  server
    .listen({ port: 4000 })
    .then(({ url }) => console.log(`GraphQL server running at ${url}`))
}

if (process.env.NODE_ENV !== 'test') {
  start()
}
```

Instead of starting the server with `server.listen()` at the top level, we put it in a function and only call it when we're not testing. However, when we run `npm test` again, while we no longer get the `console.log`, we still get the warning, which means there must be more code that we start running at the top level...

The database connection! Let's put that in a function as well:

`src/db.js`

```
import { MongoClient } from 'mongodb'

export let db

const URL = 'mongodb://localhost:27017/guide'

export const connectToDB = () => {
  const client = new MongoClient(URL, { useNewUrlParser: true })
  client.connect(e => {
    if (e) {
      console.error(`Failed to connect to MongoDB at ${URL}`, e)
      return
    }

    db = client.db()
  })
}
```

And we'll call it from `start()`. We'll also move `Github.startPolling()` from the top level of `src/data-sources/index.js`:

`src/index.js`

```
import dataSources, { Github } from './data-sources'
import { connectToDB } from './db'

const start = () => {
  connectToDB()
```

```

Github.startPolling()
server
  .listen({ port: 4000 })
  .then(({ url }) => console.log(`GraphQL server running at ${url}`))
}

if (process.env.NODE_ENV !== 'test') {
  start()
}

```

Now `npm test` completes normally. To recap, we set up integration tests for review operations by:

- Creating a test version of the server.
- Making a test utilities file that can be used like a node module.
- Writing a test.
- Mocking MongoDB collection methods.
- Preventing long-running server code from starting during testing.

Lastly, we have our second test to write— `'createReview'` :

`src/resolvers/Review.test.js`

```

test('createReview', async () => {
  const { server } = createTestServer({
    context: () => ({ user: mockUser })
  })
  const { mutate } = createTestClient(server)

  const result = await mutate({
    mutation: CREATE_REVIEW,
    variables: { review: { text: 'test', stars: 1 } }
  })
  expect(result).toMatchSnapshot()
})

```

Similarly to `'reviews'` , we create a test server and client, send an operation via the test client, and assert the response matches the snapshot. The differences are:

- We need to set the server's context as if we're logged in as `mockUser` so that we don't get the `ForbiddenError` .
- We use `mutate()` instead of `query()` , and provide the `review` variable.

For the mutation, we have:

```

const CREATE_REVIEW = gql`
  mutation CreateReview($review: CreateReviewInput!) {
    createReview(review: $review) {
      id
      text
      stars
      author {
        id
        email
      }
      createdAt
    }
  }
`

```

We include `email` , which we'll have access to because we're logged in as `mockUser` and `mockUser` will be used for the new review's `author` field.

The one thing we haven't done yet is update our database mock functions. It looks like the only new function that will be called is `this.collection.insertOne()` , which is used in `src/data-sources/Reviews.js` :

```
export default class Reviews extends MongoDataSource {
  ...

  create(review) {
    review.authorId = this.context.user._id
    review.updatedAt = new Date()
    this.collection.insertOne(review)
    return review
  }
}
```

The only thing we were depending on `insertOne()` doing was adding an `_id` property, so let's mock that:

`test/guide-test-utils.js`

```
export const createTestServer = ({ context = defaultContext } = {}) => {
  const reviews = new Reviews({
    find: jest.fn(() => ({
      toArray: jest.fn().mockResolvedValue(mockReviews)
    })),
    insertOne: jest.fn(
      doc => (doc._id = new ObjectId('5cf8b6ff37568a1fa500ba4e'))
    )
  })
  ...
}
```

Now when we run the tests, we see that two are passing, and one new snapshot is written:

```
$ npm test

> guide-api@0.1.0 test /guide-api
> jest

PASS src/resolvers/Review.test.js
  ✓ reviews (41ms)
  ✓ createReview (21ms)

  > 1 snapshot written.
Snapshot Summary
  > 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  1 written, 1 passed, 2 total
Time:       3.745s
Ran all test suites.
```

And one new snapshot is written:


`src/resolvers/__snapshots__/Review.test.js.snap`

```
exports[`createReview 1`] = `
Object {
  "data": Object {
    "createReview": Object {
      "author": Object {
        "email": "mockA@gmail.com",
        "id": "5d24f846d2f8635086e55ed3",
      },
      "createdAt": 1559803647000,
      "id": "5cf8b6ff37568a1fa500ba4e",
      "stars": 1,
      "text": "test",
    },
  },
  "errors": undefined,
}
```

```

    "extensions": undefined,
    "http": Object {
      "headers": Headers {
        Symbol(map): Object {},
      },
    },
  },
}
`
;
...

```

Looks good! 

## Code coverage

If you're jumping in here, `git checkout 21_0.2.0` (tag [21\\_0.2.0](#), or compare [21...22](#))

Jest analyzes *code coverage*—how much of our code gets run during our tests—with the `--coverage` flag. We can look at code coverage to see which parts of the code aren't covered by tests, so that we know what our new tests should cover.

Let's update our test script:

`package.json`

```

{
  ...
  "scripts": {
    ...
    "test": "jest --coverage",
    "open-coverage": "open coverage/lcov-report/index.html"
  },
}

```

When `jest --coverage` runs, it both logs statistics and updates the coverage report, which is located in the `coverage/` directory. We can now do `npm run open-coverage` for opening the HTML report. We can run `jest` without coverage with:

```
$ npx jest
```

Or to keep it open, re-running tests whenever we edit our code:

```
$ npx jest --watch
```

Or to keep it open with code coverage, one of these commands:

```

$ npx jest --coverage --watch
$ npm test -- --watch

```

`--` after an `npm` script tells `npm` to apply the subsequent arguments to the script.

We should tell `git` to ignore the generated `coverage/` report directory:

`.gitignore`

```

node_modules/
dist/
.env
coverage/

```

And we need to tell Jest which JavaScript files it should analyze coverage for, using the `collectCoverageFrom` config:

```
jest.config.js
```

```
module.exports = {
  moduleDirectories: ['node_modules', path.join(__dirname, 'test')],
  collectCoverageFrom: ['src/**/*.js']
}
```

Here's the new output of `npm test`:

```
> jest --coverage

PASS src/resolvers/Review.test.js
  ✓ reviews (49ms)
  ✓ createReview (25ms)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	34.39	21.88	35.56	33.55	
src	26.32	16.67	16.67	26.32	
context.js	28.57	25	100	28.57	... 22,25,26,27,29
db.js	25	0	0	25	8,9,10,11,12,15
formatError.js	12.5	0	0	12.5	... 12,15,16,17,19
index.js	37.5	50	0	37.5	19,20,21,23,27
src/data-sources	40.91	0	33.33	40.91	
Github.js	20	0	0	20	... 31,32,34,35,36
Reviews.js	100	100	100	100	
Users.js	33.33	100	33.33	33.33	15,16,17,21
index.js	0	100	0	0	6
src/resolvers	41.67	32.35	48	40	
Date.js	26.32	16.67	50	22.22	... 31,32,35,40,46
Github.js	0	100	0	0	6
Review.js	66.67	60	71.43	66.67	... 29,35,39,43,56
User.js	28	25	40	28	... 41,48,62,63,66
index.js	66.67	100	33.33	60	5,6
src/util	20	0	0	20	
auth.js	40	0	0	40	14,15,20,21,24,30
errors.js	0	0	0	0	... 27,28,33,35,37
pubsub.js	100	100	100	100	

```
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 2 passed, 2 total
Time: 2.641s
Ran all test suites.
```

We see the coverage overall, for each directory, and each JS file, in percentage of statements, logic branches, functions, and lines. To see which lines of code are covered, we can view the HTML report:

```
$ npm run open-coverage
```



**All files**

34.62% Statements 54/156 21.88% Branches 14/64 35.56% Functions 16/45 33.77% Lines 52/154

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File		Statements	Branches	Functions	Lines
src	<div><div></div></div>	27.03%	10/37	16.67%	3/18
src/data-sources	<div><div></div></div>	40.91%	9/22	0%	0/6
src/resolvers	<div><div></div></div>	41.67%	30/72	32.35%	11/34
src/util	<div><div></div></div>	20%	5/25	0%	0/6

And follow links to a particular file we'd like to look at, like `src/index.js` :

**All files / src index.js**

42.86% Statements 3/7 50% Branches 1/2 0% Functions 0/2 42.86% Lines 3/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```

1  import 'dotenv/config'
2  import { ApolloServer } from 'apollo-server'
3  import typeDefs from './schema/schema.graphql'
4  import resolvers from './resolvers'
5  import dataSources from './data-sources'
6  import context from './context'
7  import formatError from './formatError'
8  import { connectToDB } from './db'
9
10 1x const server = new ApolloServer({
11    typeDefs,
12    resolvers,
13    dataSources,
14    context,
15    formatError
16  })
17
18 1x const start = () => {
19    connectToDB()
20    server
21      .listen({ port: 4000 })
22      .then(({ url }) => console.log(`GraphQL server running at ${url}`))
23  }
24
25 1x if (process.env.NODE_ENV !== 'test') {
26    start()
27  }
28
29  export { ApolloServer, typeDefs, resolvers, context, formatError }
30
```

The red highlighted code shows what wasn't run during the test. Anything at the top level was run, like the imports, `ApolloServer` instance creation, and the if statement condition, but the body of the if statement—and the body of the start function, which wasn't called—wasn't run and thus is red. The highlighting isn't perfect—notice that `.listen` and `.then` should also be red but aren't.

If we want to make sure that contributors to our project write tests that cover new code, we can set a minimum coverage threshold, below which the test command will fail. We can set it for any statistic—statements, branches, functions, or lines—and either globally or for individual files. Let’s set a global statement threshold of 40%:

`jest.config.js`

```
module.exports = {
  moduleDirectories: ['node_modules', path.join(__dirname, 'test')],
  collectCoverageFrom: ['src/**/*.js'],
  coverageThreshold: {
    global: {
      statements: 40
    }
  }
}
```

Now the test will fail whenever the code coverage statement ratio is below 40%. We’re currently below 40%, so when we re-run `npm test`, it fails:

> jest --coverage

PASS

src/resolvers/Review.test.js

- ✓ reviews (38ms)
- ✓ createReview (22ms)

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	34.39	21.88	35.56	33.55	
src	26.32	16.67	16.67	26.32	
context.js	28.57	25	100	28.57	... 22,25,26,27,29
db.js	25	0	0	25	8,9,10,11,12,15
formatError.js	12.5	0	0	12.5	... 12,15,16,17,19
index.js	37.5	50	0	37.5	19,20,21,23,27
src/data-sources	40.91	0	33.33	40.91	
Github.js	20	0	0	20	... 31,32,34,35,36
Reviews.js	100	100	100	100	
Users.js	33.33	100	33.33	33.33	15,16,17,21
index.js	0	100	0	0	6
src/resolvers	41.67	32.35	48	40	
Date.js	26.32	16.67	50	22.22	... 31,32,35,40,46
Github.js	0	100	0	0	6
Review.js	66.67	60	71.43	66.67	... 29,35,39,43,56
User.js	28	25	40	28	... 41,48,62,63,66
index.js	66.67	100	33.33	60	5,6
src/util	20	0	0	20	
auth.js	40	0	0	40	14,15,20,21,24,30
errors.js	0	0	0	0	... 27,28,33,35,37
pubsub.js	100	100	100	100	

Jest: "global" coverage threshold for statements (40%) not met: 34.39%

Test Suites: 1 passed, 1 total

Tests: 2 passed, 2 total

Snapshots: 2 passed, 2 total

Time: 3.449s

Ran all test suites.

npm ERR! Test failed. See above for more details.

## User integration tests

If you’re jumping in here, `git checkout 22_0.2.0` (tag [22\\_0.2.0](#), or compare [22...23](#))

Let's try to meet our 40% coverage threshold. Looking at `src/resolvers/User.js`, we can see that our queries are red:

## All files / src/resolvers User.js

28% Statements 7/25    25% Branches 3/12    40% Functions 4/10    28% Lines 7/25

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```

1  import { AuthenticationError, ForbiddenError } from 'apollo-server'
2  import { ObjectId } from 'mongodb'
3  import { addDays, differenceInDays } from 'date-fns'
4
5  import { InputError } from '../util/errors'
6
7  const OBJECT_ID_ERROR =
8  1x  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'
9
10 export default {
11   Query: {
12     me: (_, __, context) => context.user,
13     user: (_, { id }, { dataSources }) => {
14       try {
15         return dataSources.users.findOneById(ObjectId(id))
16       } catch (error) {
17         if (error.message === OBJECT_ID_ERROR) {
18           throw new InputError({ id: 'not a valid Mongo ObjectId' })
19         } else {
20           throw error
21         }
22       }
23     },
24     searchUsers: (_, { term }, { dataSources }) =>
25       dataSources.users.search(term)
26   },

```

This makes sense, as our tests haven't sent any user queries—they've just selected `User` fields in review operations. Accordingly, when we scroll down, we see the only covered lines are for `User` field resolvers:

```

43   },
44   User: {
45     3x id: ({ _id }) => _id,
46     email(user, _, { user: currentUser }) {
47     1x   if (!currentUser || !user._id.equals(currentUser._id)) {
48       throw new ForbiddenError('cannot access others' emails')
49     }
50
51     1x   return user.email
52   },
53   photo(user) {
54     // user.authId: 'github|1615'
55     2x const githubId = user.authId.split('|')[1]
56     2x return `https://avatars.githubusercontent.com/u/${githubId}`
57   },
58   2x createdAt: user => user._id.getTimestamp()
59 },

```

Let's write some integration tests that query user operations. We'll start with the same imports and test format (one for each operation) as we did with `Review.test.js`:

`src/resolvers/User.test.js`

```
import {
```

```

    createTestServer,
    createTestClient,
    gql,
    mockUser
  } from 'guide-test-utils'

  test('me', async () => {
    ...
  })

  test('user', async () => {
    ...
  })

  test('searchUsers', async () => {
    ...
  })

  test('createUser', async () => {
    ...
  })

```

For the `me` test, we can set the `context` to a user with a certain `_id`, and then check to make sure the result's `id` matches:

```

const ME = gql`
  query {
    me {
      id
    }
  }
`

test('me', async () => {
  const { server } = createTestServer({
    context: () => ({ user: { _id: 'itme' } })
  })
  const { query } = createTestClient(server)

  const result = await query({ query: ME })
  expect(result.data.me.id).toEqual('itme')
})

```

We don't need to worry about selecting and testing other fields, as we know they've been covered.

Next is the `user` query. We know our mock users collection always returns `mockUser`, so we'll query for that user:

```

const USER = gql`
  query User($id: ID!) {
    user(id: $id) {
      id
    }
  }
`

test('user', async () => {
  const { server } = createTestServer()
  const { query } = createTestClient(server)

  const id = mockUser._id.toString()
  const result = await query({
    query: USER,
    variables: { id }
  })
  expect(result.data.user.id).toEqual(id)
})

```

For the `searchUsers` test, let's set it up so that multiple results are returned. For that, we'll need to temporarily change the mocked `users.find` function. To get access to that function, we need to get the `dataSources` from

`createTestServer()` :

```
test('searchUsers', async () => {
  const userA = { _id: 'A' }
  const userB = { _id: 'B' }
  const { server, dataSources } = createTestServer()

  dataSources.users.collection.find.mockReturnValueOnce({
    toArray: jest.fn().mockResolvedValue([userA, userB])
  })
})
```

`mockReturnValueOnce()` will have `users.find` return the given value once and then go back to returning `[mockUser]` as it was before. After we make the query, we can also test to see what `users.find` was called with:

```
expect(dataSources.users.collection.find).toHaveBeenCalledWith({
  $text: { $search: 'foo' }
})
```

All together, that's:

```
const SEARCH_USERS = gql`
  query SearchUsers($term: String!) {
    searchUsers(term: $term) {
      ... on User {
        id
      }
    }
  }
`

test('searchUsers', async () => {
  const userA = { _id: 'A' }
  const userB = { _id: 'B' }
  const { server, dataSources } = createTestServer()

  dataSources.users.collection.find.mockReturnValueOnce({
    toArray: jest.fn().mockResolvedValue([userA, userB])
  })

  const { query } = createTestClient(server)

  const result = await query({
    query: SEARCH_USERS,
    variables: { term: 'foo' }
  })

  expect(dataSources.users.collection.find).toHaveBeenCalledWith({
    $text: { $search: 'foo' }
  })
  expect(result.data.searchUsers[0].id).toEqual('A')
  expect(result.data.searchUsers[1].id).toEqual('B')
})
```

For the last test, our `createUser` mutation will be calling `users.insertOne`, which we haven't mocked yet. Let's reuse the `insertOne` function we used for reviews:

`test/guide-test-utils.js`

```
const insertOne = jest.fn(
  doc => (doc._id = new ObjectId('5cf8b6ff37568a1fa500ba4e'))
)

export const createTestServer = ({ context = defaultContext } = {}) => {
  const reviews = new Reviews({
```

```

    find: jest.fn(() => ({
      toArray: jest.fn().mockResolvedValue(mockReviews)
    })),
    insertOne
  })

  const users = new Users({
    createIndex: jest.fn(),
    find: jest.fn(() => ({
      toArray: jest.fn().mockResolvedValue(mockUsers)
    })),
    insertOne
  })

  ...

```

For the mutation input, let's pick the fields from `mockUser` :

`src/resolvers/User.test.js`

```

import { pick } from 'lodash'

const CREATE_USER = gql`
  mutation CreateUser($user: CreateUserInput!, $secretKey: String!) {
    createUser(user: $user, secretKey: $secretKey) {
      id
    }
  }
`

test('createUser', async () => {
  const { server } = createTestServer()
  const { mutate } = createTestClient(server)

  const user = pick(mockUser, [
    'firstName',
    'lastName',
    'username',
    'email',
    'authId'
  ])

  const result = await mutate({
    mutation: CREATE_USER,
    variables: {
      user,
      secretKey: process.env.SECRET_KEY
    }
  })

  expect(result).toMatchSnapshot()
})

```

Whenever we're using a snapshot, we should check it on the first run to make sure it's correct. If we run `npm test`, then we should see a new file:

`src/resolvers/__snapshots__/User.test.js.snap`

```

// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`createUser 1`] = `
Object {
  "data": Object {
    "createUser": Object {
      "id": "5cf8b6ff37568a1fa500ba4e",
    },
  },
  "errors": undefined,
  "extensions": undefined,
  "http": Object {

```

```

    "headers": Headers {
      Symbol(map): Object {},
    },
  },
}
;

```

Looks good! ✅ We can also see that our statement coverage is above the 40% minimum, so our tests pass!

```
> jest --coverage
```

```

PASS src/resolvers/Review.test.js
PASS src/resolvers/User.test.js
> 1 snapshot written.

```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	42.68	26.56	51.11	41.94	
src	26.32	16.67	16.67	26.32	
context.js	28.57	25	100	28.57	... 22,25,26,27,29
db.js	25	0	0	25	8,9,10,11,12,15
formatError.js	12.5	0	0	12.5	... 12,15,16,17,19
index.js	37.5	50	0	37.5	19,20,21,23,27
src/data-sources	59.09	0	55.56	59.09	
Github.js	20	0	0	20	... 31,32,34,35,36
Reviews.js	100	100	100	100	
Users.js	100	100	100	100	
index.js	0	100	0	0	6
src/resolvers	54.17	41.18	68	52.86	
Date.js	26.32	16.67	50	22.22	... 31,32,35,40,46
Github.js	0	100	0	0	6
Review.js	66.67	60	71.43	66.67	... 29,35,39,43,56
User.js	64	50	90	64	... 32,40,41,48,63
index.js	66.67	100	33.33	60	5,6
src/util	20	0	0	20	
auth.js	40	0	0	40	14,15,20,21,24,30
errors.js	0	0	0	0	... 27,28,33,35,37
pubsub.js	100	100	100	100	

#### Snapshot Summary

```
> 1 snapshot written from 1 test suite.
```

```
Test Suites: 2 passed, 2 total
```

```
Tests: 6 passed, 6 total
```

```
Snapshots: 1 written, 2 passed, 3 total
```

```
Time: 4.071s
```

```
Ran all test suites.
```

## Unit tests

If you're jumping in here, `git checkout 23_0.2.0` (tag [23\\_0.2.0](#), or compare [23...24](#))

We've written integration tests that cover most of our queries and mutations. If we want a higher test coverage, we could write more integration tests with different arguments or mock data that result in different parts of the code getting run. We could also write unit tests that cover individual functions. In this section we'll write a unit test that covers the `user` query. As we can see in the coverage report, we're missing three lines:



## All files / src/resolvers User.js

64% Statements 16/25 50% Branches 6/12 90% Functions 9/10 64% Lines 16/25

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```

1  import { AuthenticationError, ForbiddenError } from 'apollo-server'
2  import { ObjectId } from 'mongodb'
3  import { addDays, differenceInDays } from 'date-fns'
4
5  import { InputError } from '../util/errors'
6
7  const OBJECT_ID_ERROR =
8  2x  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'
9
10 export default {
11   Query: {
12  1x   me: (_, __, context) => context.user,
13     user: (_, { id }, { dataSources }) => {
14  1x       try {
15  1x         return dataSources.users.findOneById(ObjectId(id))
16       } catch (error) {
17         if (error.message === OBJECT_ID_ERROR) {
18           throw new InputError({ id: 'not a valid Mongo ObjectId' })
19         } else {
20           throw error
21         }
22       }
23     },

```

Let's first write a unit test that triggers the invalid ObjectId error. We can either add it to `User.test.js` or create separate files for unit tests named `File.unit.test.js`. The latter has the benefit of smaller files and we can run all the unit tests together with `npm test -- -unit`.

An alternative file structure would be to move all integration tests to the `test/` directory and only place unit tests next to the files they're testing. So `test/User.test.js` for integration and `src/resolvers/User.test.js` for unit testing `src/resolvers/User.js`.

Instead of using the test server and client, we can import the resolver function and call it ourselves:

`src/resolvers/User.unit.test.js`

```

import resolvers from './User'
import { InputError } from '../util/errors'

test('user throws InputError', () => {
  expect(() =>
    resolvers.Query.user(
      null,
      { id: 'invalid' },
      { dataSources: { users: { findOneById: jest.fn() } } }
    )
  ).toThrow(InputError)
})

```

We mock the `dataSources.users.findOneById` function, and we assert that an instance of `InputError` will be thrown.

However if we want to fit the strict definition of a unit test that says everything must be mocked, we need to mock `objectId()`. Since it's imported from an NPM module, we can use the `jest.mock()` function, which mocks the module for all the tests in the same file:

```

jest.mock('mongodb', () => ({
  ObjectId: id => {
    if (id === 'invalid') {

```



```

    throw new Error(
      'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'
    )
  }
}
})))

```

Now when `user.js` imports the function ( `import { ObjectId } from 'mongodb'` ), it will get our version of it.

For further examples of `jest.mock()`, check out the [SQL testing](#) section later on in this chapter.

When we re-run `npm test` and refresh the coverage report, we see that the statements coverage has gone up from 16/25 to 18/25:

## All files / src/resolvers User.js

**72%** Statements 18/25    **58.33%** Branches 7/12    **90%** Functions 9/10    **72%** Lines 18/25

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```

1  import { AuthenticationError, ForbiddenError } from 'apollo-server'
2  import { ObjectId } from 'mongodb'
3  import { addDays, differenceInDays } from 'date-fns'
4
5  import { InputError } from '../util/errors'
6
7  const OBJECT_ID_ERROR =
8  3x  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'
9
10 export default {
11   Query: {
12     1x  me: (_, __, context) => context.user,
13     user: (_, { id }, { dataSources }) => {
14       2x  try {
15         2x  return dataSources.users.findOneById(ObjectId(id))
16       } catch (error) {
17         1x  if (error.message === OBJECT_ID_ERROR) {
18           1x  throw new InputError({ id: 'not a valid Mongo ObjectId' })
19         } else {
20           throw error
21         }
22       }
23     },

```

There's one statement left in this function: the `throw error` line. For that, we need to have

`dataSources.users.findOneById()` throw a different error and make sure that `resolvers.Query.user()` throws the same error.

`src/resolvers/User.unit.test.js`

```

import resolvers from './User'
import { InputError } from '../util/errors'

test('user throws data source errors', () => {
  const MOCK_MONGO_ERROR = 'Unable to connect to DB'

  expect(() =>
    resolvers.Query.user(
      null,
      { id: mockMongoId },
      {
        dataSources: {
          users: {
            findOneById: () => {
              throw new Error(MOCK_MONGO_ERROR)
            }
          }
        }
      }
    )
  ).toThrow()
})

```

```

    }
  }
}
)
).toThrow(MOCK_MONGO_ERROR)
})

```

## All files / src/resolvers User.js

76% Statements 19/25 66.67% Branches 8/12 90% Functions 9/10 76% Lines 19/25

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```

1  import { AuthenticationError, ForbiddenError } from 'apollo-server'
2  import { ObjectId } from 'mongodb'
3  import { addDays, differenceInDays } from 'date-fns'
4
5  import { InputError } from '../util/errors'
6
7  const OBJECT_ID_ERROR =
8  3x  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'
9
10 export default {
11   Query: {
12 1x   me: (_, __, context) => context.user,
13     user: (_, { id }, { dataSources }) => {
14 3x     try {
15 3x       return dataSources.users.findOneById(ObjectId(id))
16     } catch (error) {
17 2x       if (error.message === OBJECT_ID_ERROR) {
18 1x         throw new InputError({ id: 'not a valid Mongo ObjectId' })
19       } else {
20 1x         throw error
21       }
22     }
23   },

```

Now the `user` query is completely green. And we could continue writing unit tests for more functions or files, either until we covered the most important pieces of logic, or until we met our overall desired test coverage percentage.

## End-to-end tests

If you're jumping in here, `git checkout 24_0.2.0` (tag [24\\_0.2.0](#), or compare [24...25](#))

The final type of testing is end-to-end, or e2e. In backend e2e testing, we start the server and database, and then we test by sending HTTP requests to the server. So our tests will look something like this:

```

beforeAll(startE2EServer)
afterAll(stopE2EServer)

test('query A', () => {
  const result = makeHttpRequest(queryA)

  expect(result).toMatchSnapshot()
})

```

Let's start by writing the `startE2EServer()` helper. We want it to look like this:

`test/guide-test-utils.js`

```

export const startE2EServer = () => {
  // start server and connect to db

```

```

    return {
      stop: () => // stops server and db client
      request: () => // send http request to server
    }
  }
}

```

It returns the `stop()` and `request()` functions for the tests to use. We can fill in the first comment:

```

import { server } from '../src/'
import { connectToDB } from '../src/db'

export const startE2EServer = async () => {
  // start server and connect to db
  const e2eServer = await server.listen({ port: 0 })
  await connectToDB()

  return {
    stop: () => // stops server and db client
    request: () => // send http request to server
  }
}

```

`{ port: 0 }` uses any available port, which we do because the default port (4000) will be in use if our dev server is running while we run our tests. In order to `await` our call to `connectToDB()`, we need to make it async instead of callback-based:

`src/db.js`

```

export let db

export const connectToDB = async () => {
  const client = new MongoClient(URL, { useNewUrlParser: true })
  await client.connect()
  db = client.db()
  return client
}

```

We also need to `return client` so that we can close the connection when testing is done. For stopping the server, there is a `e2eServer.server.close`, but it's callback-based. We can use node's `promisify()` to turn it into a Promise that we can `await`:

`test/guide-test-utils.js`

```

import { promisify } from 'util'

export const startE2EServer = async () => {
  const e2eServer = await server.listen({ port: 0 })
  const dbClient = await connectToDB()

  const stopServer = promisify(e2eServer.server.close.bind(e2eServer.server))

  return {
    stop: async () => {
      await stopServer()
      await dbClient.close()
    }
    request: () => // send http request to server
  }
}

```

We also use `bind` to maintain the function's `this`.

We can make our function run faster by performing startup and stopping in parallel using `Promise.all()`:

```
export const startE2EServer = async () => {
  const [e2eServer, dbClient] = await Promise.all([
    server.listen({ port: 0 }),
    connectToDB()
  ])

  const stopServer = promisify(e2eServer.server.close.bind(e2eServer.server))

  return {
    stop: () => Promise.all([stopServer(), dbClient.close()]),
    request: () => // send http request to server
  }
}
```

Lastly, we can send HTTP requests to the server using Apollo Link. `apollo-link-http` has the basic `HttpLink` and `apollo-link` has `execute()`, a function that sends GraphQL operations over a link, and `toPromise()`, which converts the Observable that `execute()` returns into a Promise. All together, that's:

`test/guide-test-utils.js`

```
import { promisify } from 'util'
import { HttpLink } from 'apollo-link-http'
import fetch from 'node-fetch'
import { execute, toPromise } from 'apollo-link'

import { server } from '../src/'
import { connectToDB } from '../src/db'

export const startE2EServer = async () => {
  const [e2eServer, dbClient] = await Promise.all([
    server.listen({ port: 0 }),
    connectToDB()
  ])

  const stopServer = promisify(e2eServer.server.close.bind(e2eServer.server))

  const link = new HttpLink({
    uri: e2eServer.url,
    fetch
  })

  return {
    stop: () => Promise.all([stopServer(), dbClient.close()]),
    request: operation => toPromise(execute(link, operation))
  }
}
```

We also need `src/index.js` to add `server` to its exports:

`src/index.js`

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError
})

...

export { server, typeDefs, resolvers, context, formatError }
```

Now we can write our e2e test:

`test/e2e.test.js`

```
import { gql, startE2EServer } from 'guide-test-utils'

let stop, request

beforeAll(async () => {
  const server = await startE2EServer()
  stop = server.stop
  request = server.request
})

afterAll(() => stop())

const HELLO = gql`
  query {
    hello
  }
`

test('hello', async () => {
  const result = await request({ query: HELLO })

  expect(result).toMatchSnapshot()
})
```

We start the server in `beforeAll()` and stop it in `afterAll()`. Then we create our query document, which we send to the server using `request()` in our one test. After we run the test, we check the snapshot:

`test/__snapshots__/e2e.test.js.snap`

```
exports[`hello 1`] = `
Object {
  "data": Object {
    "hello": "      ",
  },
}
`;
```

## Production

Once our GraphQL server code works and we want others (or our client code) to use it, we need to put it into production. This section contains the basic steps:

- [Deploying our code](#)
- [Setting up our databases](#)
- [Gathering analytics](#)
- [Tracking errors](#)

And in the last part of the chapter, [Extended topics](#), we cover additional topics relevant to production, including [security](#) and [performance](#).

## Deployment

- [Options](#)
- [Deploying](#)
- [Environment variables](#)

### Options

For our GraphQL API to be accessible, we need our code to run on a server that is *publicly addressable*—i.e., it can be reached via a public IP address. Our dev computer usually can't be reached because it has a local (non-public) IP address (often starting with `192.168.*.*`), and the router that connects us to the internet (which does have a public IP) usually doesn't respond to HTTP requests. While we could set the router up to forward requests to our dev computer, we then would have to leave our computer there and powered on, as well as do a number of other things to keep it working (like [DDNS](#)). Given the trouble and unreliability of that solution, we usually run our server code on a different computer—a production server—that's been built, set up, and maintained for that purpose.

The *deployment* process is copying the latest version of our code to the production server and running it. There are four main types of production servers we can use:

- **On-prem:** In *on premises*, we buy our own server, plug it into a power outlet, connect it to the internet, and then maintain it ourselves.
- **IaaS:** In *infrastructure as a service*, a company (like Amazon with its EC2 service) houses and maintains the physical servers, and we choose the operating system. We connect to the operating system over SSH to get a command prompt and then install Node, copy our code to the machine, and run it.
- **PaaS:** *Platform as a service* is like IaaS, except in addition to maintaining the physical servers, the company also maintains the operating system and software server. For example, a Node PaaS company would install and update Node.js, and we would send them our code, and they would run it with their version of Node.
- **FaaS:** *Function as a service* (also known as *serverless*) is like PaaS, except instead of sending them Node server code (which runs continuously and responds to any path / route), we send them individual JavaScript functions and configure which route triggers which function. Then, when we get HTTP requests, their server runs the right function. The function returns the response, which their server forwards to the client. Once the function returns, our code stops running—with FaaS, we don't have a continuously running server process.

These options appear in:

- decreasing order of complexity to use. It's most difficult to run our own server, and it's easiest to write and upload a single function.
- increasing time order:
  - 1970s: On-prem was the original type of server since the beginning of the internet.
  - 2006: Amazon Web Services (AWS) came out with EC2, the most popular IaaS.
  - 2009: Heroku, which popularized PaaS, publicly launched.
  - 2014: AWS came out with Lambda, the most popular FaaS.

Currently, PaaS seems to be the most popular option in modern web development. However, FaaS is rising and may eclipse PaaS. Notably, the most popular PaaS in the Node community ([Vercel Now](#), formerly Zeit Now), switched to FaaS. While FaaS might be better for many applications, there are some disadvantages:

- **No continuous server process:** When we have a process (as we do with on-prem, IaaS, and PaaS), we can do things like:
  - Store data in memory between requests. The alternative that usually suffices is using an independent memory store, like a Redis server, which adds a small network latency (only ~0.2ms if it's inside the same AWS Availability Zone).
  - Open and maintain a WebSocket connection. However, some FaaS providers have added the ability to use WebSockets: At the end of 2018, AWS added support for WebSockets to its API Gateway, which can call a Lambda function when each message arrives over the socket.
- **Database limitations:** Since there's no continuous server process, our database client library can't maintain a pool of connections for our requests to go out on; instead, each function makes its own connection. So the database has to be able to accept many connections over SSL.
- **Latency:** When there's not an existing server process, the FaaS provider has to start a new process (with a copy of our code and npm packages) to handle an incoming request, and that takes time, which increases the latency (i.e., total response time of the server). For example, Lambda usually takes under 500ms to create a new instance to handle a request (also called a *cold start*). Once the function returns, the instance continues running

and immediately handles the next request that arrives. If there are no requests for about ten seconds, it shuts down, and the next request is subject to the 500ms instance startup latency. Also, if there's an existing instance handling a request and a second request arrives while the existing instance is busy, a second instance is cold-started.

- **Resource limits:** FaaS providers usually limit how much memory and CPU can be used and how long the function can run. One of the more flexible providers is Lambda. By default, it limits memory and duration to 128 MB and 3 seconds. The limits can be raised to a maximum 3,008 MB and 15 minutes, which costs more. CPU speed scales linearly with memory size.

An example of an application that isn't well-suited to FaaS is a [Meteor](#) app, which:

- Keeps a WebSocket open to every client.
- Stores in memory a cache of each client's data.
- Can use a lot of CPU to determine what data updates to send to each client.

Apollo Server [doesn't yet support](#) GraphQL subscriptions on Lambda. [aws-lambda-graphql](#) is a different GraphQL server that does support subscriptions on Lambda. Aside from subscriptions, FaaS is a great fit for GraphQL because:

- GraphQL only has a single route, so we only need one function.
- The only thing stored in memory between requests is the data source cache, and that's easy to swap out with a Redis cache.

Since our app uses subscriptions, let's use Heroku, a PaaS that supports Node.

It's worth noting that another option would be to split our application layer between two servers:

- One that handles Queries and Mutations over HTTP, hosted on a FaaS.
- One that handles Subscriptions over WebSockets, hosted on a PaaS.

The former could publish subscription events to Redis, which the latter could subscribe to.

## Deploying

If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...26](#))

In this section we'll deploy our server to the Heroku PaaS, see how it breaks, and then fix it 🤖.

We start by [creating an account](#). Then we do the following steps:

```
$ cd guide-api/
$ brew install heroku/brew/heroku
$ heroku login
$ heroku create
$ git push heroku 25:master
$ heroku open
```

1. `brew install heroku/brew/heroku` —Install the `heroku` command-line tool.
2. `heroku login` —Log in using the account we just created.
3. `heroku create` —Create a new Heroku app. This registers our server with Heroku and reserves a name (which is used in the deployed URL: `https://app-name.herokuapp.com/`). It also adds a Git remote named `heroku`.
4. `git push heroku 25:master` —Git push to the master branch of the Heroku remote. When Heroku receives the updated code, it builds and runs the server. This command assumes we currently have branch 25 checked out on our machine. If we were on `master`, we could just run `git push heroku master`.
5. `heroku open` —Open the deployed URL in the browser.

On the page that's opened (`https://app-name.herokuapp.com/`), we see "Application error," which we can investigate by viewing the logs:

```
$ heroku logs
```

This prints a lot of logs, including:

```
2019-10-30T12:50:33.923678+00:00 heroku[web.1]: Error R10 (Boot timeout) -> Web process failed to bind to $PORT
within 60 seconds of launch
2019-10-30T12:50:33.951435+00:00 heroku[web.1]: Stopping process with SIGKILL
```

When Heroku runs our code, it provides a `PORT` environment variable and waits for our code to start a server on that port. If our code doesn't do so within a minute, Heroku kills the process. We're running our server on port 4000, so it killed us. 🤖😞

To resolve this problem, let's update our code to use `PORT` :

```
src/index.js
```

```
server
  .listen({ port: process.env.PORT || 4000 })
  .then(({ url }) => console.log(`GraphQL server running at ${url}`))
```

We fall back to `4000` in development, where there is no `PORT` environment variable. Now to test, we can run `heroku logs --tail` in one terminal ( `--tail` keeps the command running, displaying log lines in real time) and deploy in another. Since the deployment process for Heroku is `git push`, we have to create a new commit, so that the updated code is part of the push.

```
$ git add src/index.js
$ git commit -m 'Listen on process.env.PORT in production'
$ git push heroku 25:master
```

After the last command, we should start seeing log lines like this (plus timestamps) in the first terminal:

```
$ heroku logs --tail
...
app[api]: Build started by user loren@graphql.guide
heroku[web.1]: State changed from crashed to starting
app[api]: Release v4 created by user loren@graphql.guide
app[api]: Deploy 4f2d2e92 by user loren@graphql.guide
app[api]: Build succeeded
heroku[web.1]: Starting process with command `npm start`
app[web.1]:
app[web.1]: > guide-api@0.1.0 start /app
app[web.1]: > node dist/index.js
app[web.1]:
app[web.1]: GraphQL server running at http://localhost:7668/
app[web.1]: (node:23) UnhandledPromiseRejectionWarning: MongoNetworkError: failed to connect to server [localhost:27017] on first connect [Error: connect ECONNREFUSED 127.0.0.1:27017]
heroku[web.1]: State changed from starting to up
app[web.1]: Error: GraphQL Error (Code: 401): {"response":{"message":"Bad credentials","documentation_url":"https://developer.github.com/v4","status":401},"request":{"query":"\nquery GuideStars {\n  repository(owner: \"GraphQLGuide\", name: \"guide\") {\n    stargazers {\n      totalCount\n    }\n  }\n}\n"}}
heroku[web.1]: State changed from up to crashed
```

Heroku didn't kill us! 🎉👏👏

We can kill the logs process by hitting `ctrl-C`.

The label `[web.1]` identifies which *dyno* (Heroku's term for a container) the log comes from. By default, our app only has one dyno, but we could scale up to multiple if we wanted. The lines labeled `heroku` are the dyno's general state changes:

```
heroku[web.1]: State changed from crashed to starting
```



```
heroku[web.1]: Starting process with command `npm start`
heroku[web.1]: State changed from starting to up
```

The lines labeled `app` are more granular and include all the output from our server process. The last two lines are errors that we'll fix in the next two sections:

```
app[web.1]: (node:23) UnhandledPromiseRejectionWarning: MongoNetworkError: failed to connect to server [localhost:27017] on first connect [Error: connect ECONNREFUSED 127.0.0.1:27017]

app[web.1]: Error: GraphQL Error (Code: 401): {"response":{"message":"Bad credentials","documentation_url":"https://developer.github.com/v4","status":401},"request":{"query":"\nquery GuideStars {\n  repository(owner: \"GraphQLGuide\", name: \"guide\") {\n    stargazers {\n      totalCount\n    }\n  }\n}\n\"}}}
```

## Environment variables

If you're jumping in here, `git checkout 26_0.1.0` (tag [26\\_0.1.0](#)).

There are a couple outstanding errors with our deployment. Let's look at this one:

```
app[web.1]: Error: GraphQL Error (Code: 401): {"response":{"message":"Bad credentials","documentation_url":"https://developer.github.com/v4","status":401},"request":{"query":"\nquery GuideStars {\n  repository(owner: \"GraphQLGuide\", name: \"guide\") {\n    stargazers {\n      totalCount\n    }\n  }\n}\n\"}}}
```

It's an error response from our `GuideStars` query which our server is sending to GitHub's API. The error message is `Bad credentials`. Credentials are provided in the authorization header:

`src/data-sources/Github.js`

```
const githubAPI = new GraphQLClient('https://api.github.com/graphql', {
  headers: {
    authorization: `bearer ${process.env.GITHUB_TOKEN}`
  }
})
```

The problem is the `GITHUB_TOKEN` *environment variable* (env var) isn't defined, because our `.env` file isn't in Git, which means Heroku didn't get a copy of the file when we did `git push`. To set environment variables, PaaS and FaaS providers have a web UI and/or command-line tool. Heroku has both—let's fix our problem with its command-line tool:

```
$ heroku config:set GITHUB_TOKEN=...
Setting GITHUB_TOKEN and restarting ● graphql-guide... done, v5
GITHUB_TOKEN: ...
```

Replace `...` with the value from our `.env` file.

Then, Heroku restarts the server to provide the new environment variable. We can now see with `heroku logs` that the `Bad credentials` error doesn't appear after the restart.

We need to also set our other environment variable from `.env`:

```
$ heroku config:set SECRET_KEY=...
```

## Database hosting

- [MongoDB hosting](#)
- [Redis hosting](#)

- [Redis PubSub](#)
- [Redis caching](#)

## MongoDB hosting

If you're jumping in here, `git checkout 26_0.1.0` (tag [26\\_0.1.0](#), or compare [26...27](#))

Our last error is:

```
app[web.1]: (node:23) UnhandledPromiseRejectionWarning: MongoNetworkError: failed to connect to server [localho
st:27017] on first connect [Error: connect ECONNREFUSED 127.0.0.1:27017]
```

The error is coming from MongoDB, which we're setting up with:

`src/db.js`

```
const URL = 'mongodb://localhost:27017/guide'

export const connectToDB = async () => {
  const client = new MongoClient(URL, { useNewUrlParser: true })
  await client.connect()
  db = client.db()
  return client
}
```

In production, `localhost` is our Heroku container, which doesn't have a MongoDB database server running on it. We need a place to host our database, and then we can use that URL instead of `mongodb://localhost:27017/guide`.

We have similar options to our Node [deployment options](#): on-prem, IaaS, and DBaaS (similar to PaaS). Most people choose DBaaS because it requires the least amount of effort. With on-prem, we'd have to house the machines, and with IaaS, we'd have to configure and manage the OS and database software ourselves. MongoDB, Inc. runs their own DBaaS called [Atlas](#).

Let's use the Atlas free plan to get a production MongoDB server. During setup, we have a choice of which cloud provider we want our database to be hosted on: AWS, Google Cloud Platform, or Microsoft Azure. Within the cloud provider, we also need to choose a region:

## Cloud Provider &amp; Region

AWS, N. Virginia (us-east-1) ▾



Create a **free tier cluster** by selecting a region with **FREE TIER AVAILABLE** and choosing the **M0** cluster tier below.

★ Recommended region ⓘ

NORTH AMERICA	EUROPE	ASIA
<div>  N. Virginia (us-east-1) ★ FREE TIER AVAILABLE         </div> <div>  Ohio (us-east-2) ★         </div> <div>  N. California (us-west-1)         </div> <div>  Oregon (us-west-2) ★ FREE TIER AVAILABLE         </div> <div>  Montreal (ca-central-1)         </div>	<div>  Stockholm (eu-north-1) ★         </div> <div>  Ireland (eu-west-1) ★ FREE TIER AVAILABLE         </div> <div>  London (eu-west-2) ★         </div> <div>  Paris (eu-west-3) ★         </div> <div>  Frankfurt (eu-central-1) ★ FREE TIER AVAILABLE         </div>	<div>  Hong Kong (ap-east-1) ★         </div> <div>  Tokyo (ap-northeast-1) ★         </div> <div>  Seoul (ap-northeast-2)         </div> <div>  Singapore (ap-southeast-1) ★ FREE TIER AVAILABLE         </div> <div>  Mumbai (ap-south-1) FREE TIER AVAILABLE         </div>
SOUTH AMERICA	AUSTRALIA	MIDDLE EAST
<div>  Sao Paulo (sa-east-1)         </div>	<div>  Sydney (ap-southeast-2) ★ FREE TIER AVAILABLE         </div>	<div>  Bahrain (me-south-1) ★         </div>

As discussed in the [Latency](#) background section, we want to pick the provider and region closest to our Heroku GraphQL server so that our GraphQL server can reach the database quickly.

Here are all the Heroku regions:

\$ heroku regions		
ID	Location	Runtime
eu	Europe	Common Runtime
us	United States	Common Runtime
dublin	Dublin, Ireland	Private Spaces
frankfurt	Frankfurt, Germany	Private Spaces
oregon	Oregon, United States	Private Spaces
sydney	Sydney, Australia	Private Spaces
tokyo	Tokyo, Japan	Private Spaces
virginia	Virginia, United States	Private Spaces

Our server is in the default region, `us`. We can look up more information about `us` using Heroku's API:

```
$ curl -n -X GET https://api.heroku.com/regions/us -H "Accept: application/vnd.heroku+json; version=3"
{
  "country": "United States",
  "created_at": "2012-11-21T20:44:16Z",
  "description": "United States",
  "id": "59accabd-516d-4f0e-83e6-6e3757701145",
  "locale": "Virginia",
  "name": "us",
  "private_capable": false,
```

```

"provider":{
  "name":"amazon-web-services",
  "region":"us-east-1"
},
"updated_at":"2016-08-09T22:03:28Z"
}

```

Under the `provider` attribute, we can see that the Heroku `us` region is hosted on AWS's `us-east-1` region. So let's pick `AWS` and `us-east-1` for our Atlas database hosting location. Now it will take less than a millisecond for our GraphQL server to talk to our database.

After a few minutes, our cluster has been created, and we can click the "Connect" button:

**mongoDB Atlas** All Clusters

**CONTEXT**

guide-api

**SELF > GUIDE-API**

## Clusters

Find a cluster...

**ATLAS**

**Clusters**

Data Lake BETA

**SECURITY**

Database Access

Network Access

Advanced

**PROJECT**

Access Management

Activity Feed

Alerts 0

Integrations

Settings

**SANDBOX**

**Cluster0**

Version 4.0.13

**CONNECT** **METRICS** **COLLECTIONS** **...**

**CLUSTER TIER**

M0 Sandbox (General)

**REGION**

AWS / N. Virginia (us-east-1)

**TYPE**

Replica Set - 3 nodes

**LINKED STITCH APP**

None Linked

The first step is "Whitelist your connection IP address." IP *safelisting* (formerly known as "whitelisting") only allows certain IP addresses to connect to the database. The IP address we want to be able to connect to the database is the IP of our GraphQL server. However, our Heroku dynos have different IPs, and the `IPs of us-east-1` change over time. And, even if they were static, it wouldn't be very secure to list them, as an attacker could rent a machine in `us-`

east-1 to run their code on. As an alternative, we could use a [Heroku add-on](#) to provide a static outbound IP address for all of our dynos, but, for now let's go the easy and less secure route of safelisting all IP addresses. Use `0.0.0.0/0` to denote the range of all addresses.

This issue isn't specific to Heroku or MongoDB—it applies to any database that's used by any server platform with shared IP addresses.

Next we'll create a username and password. On the “Choose a connection method” step, we choose “Connect your application” and copy the connection string, which looks like this:

```
mongodbsrv://<username>:<password>@cluster0-9ofk6.mongodb.net/test?retryWrites=true&w=majority
```

The `cluster0-*****.mongodb.net` domain is the domain of our new MongoDB cluster, which can contain multiple databases. The `/test?` part determines the default database. Let's change ours to `/guide?`. We also need to replace `<username>` and `<password>` with the user we created.

Then we can set our URL as an environment variable:

```
$ heroku config:set MONGO_URL="mongodbsrv://***:***@cluster0-*****.mongodb.net/guide?retryWrites=true&w=majority"
```

And finally, we can reference it in the code:

`src/db.js`

```
const URL = process.env.MONGO_URL || 'mongodb://localhost:27017/guide'
```

At this point, our new database is empty. We can either recreate our user document using Compass or run this command to copy all our users and reviews from our local database to the production database:

```
mongodump --archive --uri "mongodb://localhost:27017/guide" | mongorestore --archive --uri "mongodbsrv://..."
```

Replace `mongodbsrv://...` with your URL.

After we commit and push to Heroku, we can see our server is error-free! 🐛

```
$ heroku logs
heroku[web.1]: Starting process with command `npm start`
app[web.1]:
app[web.1]: > guide-api@0.1.0 start /app
app[web.1]: > node dist/index.js
app[web.1]:
app[web.1]: GraphQL server running at http://localhost:33029/
heroku[web.1]: State changed from starting to up
```

## Redis hosting

Background: [Redis](#)

If you're jumping in here, `git checkout 27_0.1.0` (tag [27\\_0.1.0](#), or compare [27...28](#))

There are two parts of our app that are only meant to run in development, and we need to change for production:

- Apollo Server's included `PubSub` implementation, which we use for subscriptions.
- Apollo Server's default cache, which is used by data sources.

Both of these things were designed to work when the server runs as a single continuous process. In production, there are usually multiple processes/containers/servers, PaaS containers are subject to being restarted, and FaaS definitely isn't continuous 😞.

To get ready for production, let's use a `PubSub` implementation and cache library that were designed for [Redis](#), the most popular caching (in-memory) database.

## Redis PubSub

Our current `PubSub` comes from `apollo-server` :

`src/util/pubsub.js`

```
import { PubSub } from 'apollo-server'

export const pubsub = new PubSub()
```

There are many `PubSub` implementations for different databases and queues (see [Apollo docs > Subscriptions > PubSub Implementations](#)). We'll use `RedisPubSub` from `graphql-redis-subscriptions` when we're in production:

```
import { PubSub } from 'apollo-server'
import { RedisPubSub } from 'graphql-redis-subscriptions'

import { getRedisClient } from './redis'

const inProduction = process.env.NODE_ENV === 'production'

const productionPubSub = () => new RedisPubSub({
  publisher: getRedisClient(),
  subscriber: getRedisClient()
})

export const pubsub = inProduction ? productionPubSub() : new PubSub()
```

We have the same line checking `NODE_ENV` in `formatError.js`, so let's deduplicate by adding a new file:

`src/env.js`

```
export const inProduction = process.env.NODE_ENV === 'production'
```

`src/formatError.js`

```
import { inProduction } from './env'
```

`src/util/pubsub.js`

```
import { inProduction } from '../env'
```

The one piece we haven't seen yet is `getRedisClient` :

`src/util/redis.js`

```
import Redis from 'ioredis'

const { REDIS_HOST, REDIS_PORT, REDIS_PASSWORD } = process.env

const options = {
  host: REDIS_HOST,
  port: REDIS_PORT,
  password: REDIS_PASSWORD,
  retryStrategy: times => Math.min(times * 50, 1000)
}

export const getRedisClient = () => new Redis(options)
```

We use our preferred Redis client library, `ioredis`. The `retryStrategy` function returns how long to wait (in milliseconds) before trying to reconnect to the server when the connection is broken.

We need a public Redis server to connect to. For that, we'll use Redis Labs, the sponsor of Redis. They have a DBaaS, and it includes a [free 30MB tier](#) we can use. During sign-up, we have to choose a cloud provider and region (we'll use AWS and us-east-1, since that's where our GraphQL server is hosted), as well as an eviction policy:

`allkeys-lfu`. An eviction policy determines which keys get deleted when the 30MB of memory is full, and `lfu` stands for least frequently used.

Once we've signed up, we'll have connection info like this:

`.env`

```
REDIS_HOST=redis-10042.c12.us-east-1-4.ec2.cloud.redislabs.com
REDIS_PORT=10042
REDIS_PASSWORD=abracadabra
```

Once the info is added to our `.env` file, our `getRedisClient()` function (and our pubsub system) should start working.

We can check to make sure it's connecting to the right Redis server by turning on debug output: in the `dev` script in our `package.json`, add `DEBUG=ioredis:*` before `babel-node src/index.js`.

We can also test our new Redis-backed pubsub by making a subscription in Playground, unstarring and starring the repo [on GitHub](#), and confirming that two events appear:

```
1 subscription {
2   githubStars
3 }
```

```
{
  "data": {
    "githubStars": 86
  }
}
```

```
{
  "data": {
    "githubStars": 87
  }
}
```

Listening ...

## Redis caching

Apollo Server's default cache for data sources is an in-memory LRU cache (*LRU* means that when the cache is full, the *least recently used* data gets evicted). To ensure our data source classes across multiple containers have the same cached data, we'll switch to a Redis cache. The `'apollo-server-cache-redis'` library provides `RedisCache` :

`src/util/redis.js`

```
import Redis from 'ioredis'
import { RedisCache } from 'apollo-server-cache-redis'

const { REDIS_HOST, REDIS_PORT, REDIS_PASSWORD } = process.env

const options = {
  host: REDIS_HOST,
  port: REDIS_PORT,
  password: REDIS_PASSWORD,
  retryStrategy: times => Math.min(times * 50, 1000)
}

export const getRedisClient = () => new Redis(options)

export const cache = new RedisCache(options)

export const USER_TTL = { ttl: 60 * 60 } // hour
```

We added the `cache` and `USER_TTL` exports. Now we can add `cache` to the `ApolloServer` constructor:

`src/index.js`

```
import { cache } from '../util/redis'

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError,
  cache
})
```

To use caching, we have to set a *TTL* (time to live) with our calls to `findOneById` . This argument denotes how many seconds an object will be kept in the cache, during which calls to `findOneById` with the same ID will return the cached object instead of querying the database.

We choose a TTL based on our app requirements and how often our objects change. Our user documents rarely change, and it wouldn't be a big deal for one to be less than an hour out of date after a change, so we can set the TTL for user documents to an hour (60 \* 60 seconds). We're not currently using `findOneById` for reviews, but if we did, we might use a lower TTL—maybe a minute—if we want users to be able to edit their reviews and see those changes reflected in the app sooner.

Now let's add `USER_TTL` to our `User` and `Review` resolvers:

`src/resolvers/User.js`

```
import { USER_TTL } from '../util/redis'

export default {
  Query: {
    me: ...
    user: (_, { id }, { dataSources }) => {
      try {
        return dataSources.users.findOneById(ObjectId(id), USER_TTL)
      } catch (error) {
        if (error.message === OBJECT_ID_ERROR) {
```



```

        throw new InputError({ id: 'not a valid Mongo ObjectId' })
      } else {
        throw error
      }
    },
    searchUsers: ...
  },
  ...
}

```

src/resolvers/Review.js

```

import { USER_TTL } from '../util/redis'

export default {
  Query: {
    reviews: ...
  },
  Review: {
    id: ...
    author: (review, _, { dataSources }) =>
      dataSources.users.findOneById(review.authorId, USER_TTL),
    fullReview: async (review, _, { dataSources }) => {
      const author = await dataSources.users.findOneById(
        review.authorId,
        USER_TTL
      )
      return `${author.firstName} ${author.lastName} gave ${review.stars} stars, saying: "${review.text}"`
    },
    createdAt: ...
  },
  ...
}

```

Now after we make a query like `{ reviews { fullReview } }`, we should be able to see a user object stored in Redis. To view the database's contents, we can use the command line ( `brew install redis` and then `redis-cli -h` ) or a GUI like [Medis](#):

type	name
STR	mongo-users-5d24f846d2f8635086e55ed3

```

1 {
2   "_id": "5d24f846d2f8635086e55ed3",
3   "firstName": "John",
4   "lastName": "Resig",
5   "username": "jeresig",
6   "email": "john@graphql.guide",
7   "authId": "github|251288",
8   "updatedAt": "2019-06-05T21:24:41.889Z"
9 }

```

Save Changes

Keys: 1 DB: 0 (1) Bytes: 186 Encoding: raw TTL: 59.91min

The cache key has the format `mongo-[collection]-[id]`, and the value is a string, formatted by Medis as JSON. We can also see the remaining TTL on the bottom right.

Finally, let's get Redis working in production. We update our environment variables on Heroku with:

```

$ heroku config:set \
  REDIS_HOST=redis-10042.c12.us-east-1-4.ec2.cloud.redislabs.com \
  REDIS_PORT=10042 \

```

```
REDIS_PASSWORD=abracadabra
```

And we push the latest code:

```
$ git commit -am 'Add Redis pubsub and caching'
$ git push heroku 27:master
```

We'll learn in the next section how to query our production API. For now, we can test our Redis in production by deleting the `mongo-users-foo` key, making the same `{ reviews { fullReview } }` query, and then refreshing Medis to ensure the key has been recreated.

## Querying in production

If you're jumping in here, `git checkout 28_0.1.0` (tag `28_0.1.0`).

Now when we visit our `app-name.herokuapp.com`, instead of "Application error" we see:

```
GET query missing.
```

Usually GraphQL requests are sent by POST, but Apollo Server also supports receiving GET requests. The browser is making a `GET /` request when we load the page, but the format that Apollo supports is `GET /?query=X`. Let's test it with the `{ hello }` query:

```
app-name.herokuapp.com/?query={hello}
```

A screenshot of a web browser's address bar. It shows navigation icons (back, forward, refresh) on the left, a lock icon, and the URL `graphql-guide.herokuapp.com/?query={hello}`.

```
{
  - data: {
    hello: "🌍🌍🌍"
  }
}
```

This method of querying our production server works, but it becomes inconvenient if queries are large or use variables, and we can't add an authorization header. The method we were using before, GraphQL Playground, is disabled by default in production. However, we can use the [Playground app](#) (download the latest `.dmg` or `.exe` file) to query any GraphQL API. First, we select "URL ENDPOINT" and enter our production URL:

# New Workspace

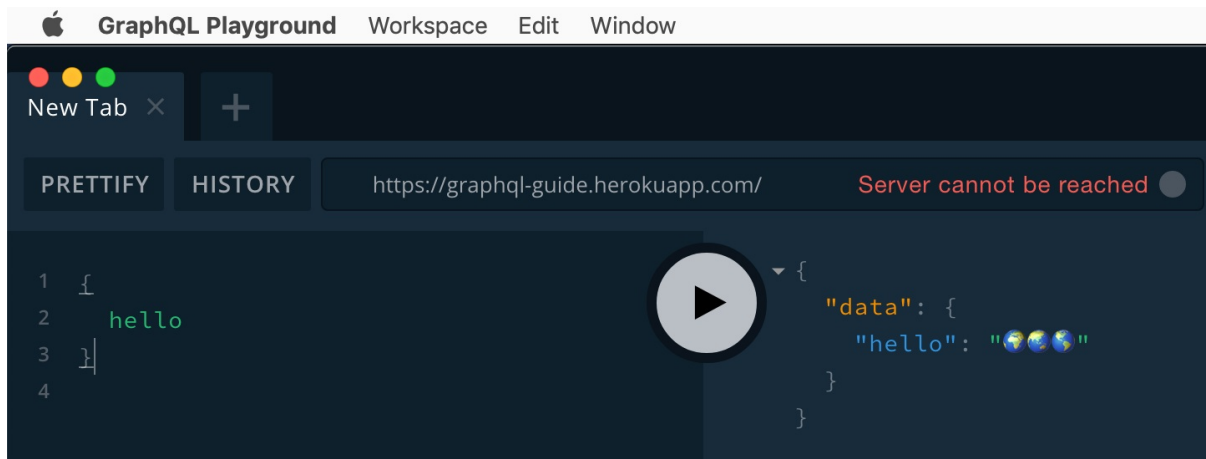
Either load a local repository with a  
.graphqlconfig file, or just open a  
HTTP endpoint

LOCAL

URL ENDPOINT



And then we query:



While the query returns a response, we see the “Server cannot be reached” error. Query autocompletion doesn’t work, and the schema tab doesn’t load. These issues occur because *introspection*—the queries that return the schema—is disabled by default in production in order to obscure private APIs.

*Private* APIs are meant to be used only by the company’s own applications, versus public APIs like the [GitHub API](#) that are meant to be used by third parties.

If we were publishing a public API that we wanted third-party apps to query, we would want to enable at least introspection (and probably Playground as well) to make it easier for the third-party developers to query our API.

We can enable both introspection and Playground in production by adding the last two options below:

src/index.js

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError,
  introspection: true,
  playground: true
})
```

```
$ git add src/index.js
$ git commit -m 'Enable introspection and Playground'
$ git push heroku 26:master
```

Now we can view the schema in the Playground app, and if we visit our index URL, the Playground website will load:

[app-name.herokuapp.com/](http://app-name.herokuapp.com/)

If we want to undo the change, we can do:

```
$ git reset HEAD^
$ git checkout -- src/index.js
$ git push heroku 26:master -f
```

We need the `-f` (force push). A normal push will fail because the `heroku` remote's `master` branch is in a different state from our branch `26` (`heroku` still has the “Enable introspection and Playground” commit as the branch tip).

In summary, the ways we can interactively query our production GraphQL server are:

- `GET /?query=X`
- Playground app without introspection
- Playground app with introspection (the server must have introspection enabled)
- Playground website, if the server has it enabled

And we can, of course, continue to query it with POST requests on the command line or in code.

## Analytics

If you're jumping in here, `git checkout 28_0.1.0` (tag [28\\_0.1.0](#)).

There are different types of server analytics that can be useful to track and a variety of tools that gather and display data. In this section, we'll cover a few of each.

The types of analytics can be split into two categories: those at the operating system (OS) level, and those at the code level.

At the OS level, there are:

- Memory usage
- CPU usage
- Request statistics, which include:
  - Rate (e.g. 1000 requests received per second)
  - Response times (e.g. 95% of responses are sent within 100ms of receiving the request)
  - Error rates (e.g., 1% of responses have an HTTP code in the 500-599 range)

The code level can also measure things based on details in the code: for instance, [Express](#) route names or GraphQL field usage. A popular *application performance management* (APM) tool that can do code-level measurement is [New Relic], which has an npm library that tracks requests by route name for a list of [supported frameworks](#) like Express. It also can monitor the performance of calls to several different databases, and it provides functions for custom instrumentation/metrics.

The main APM tool for GraphQL servers is Apollo's Graph Manager, which tracks the request statistics listed above, as well as:

- Queries received
- Fields selected
- Resolver timelines

- Clients
- Deprecated field usage
- GraphQL errors

For memory and CPU usage, we could either use Heroku's [built-in metrics](#) or New Relic. However, these OS-level metrics are becoming less important, given the prevalence of autoscaling (where the PaaS automatically adds more containers when under a high load) and serverless (where we usually don't have to think about memory and CPU).

For the rest of the metrics, let's set up Graph Manager. First, we [sign up](#), and then we're given an API key to set for the `ENGINE_API_KEY` env variable:

```
.env
```

```
ENGINE_API_KEY=service:guide-api:*****
```

We now start our server with `npm run dev`. Once it has finished starting up, we run this command in a new terminal:

```
$ npx apollo service:push --endpoint="http://localhost:4000"
```

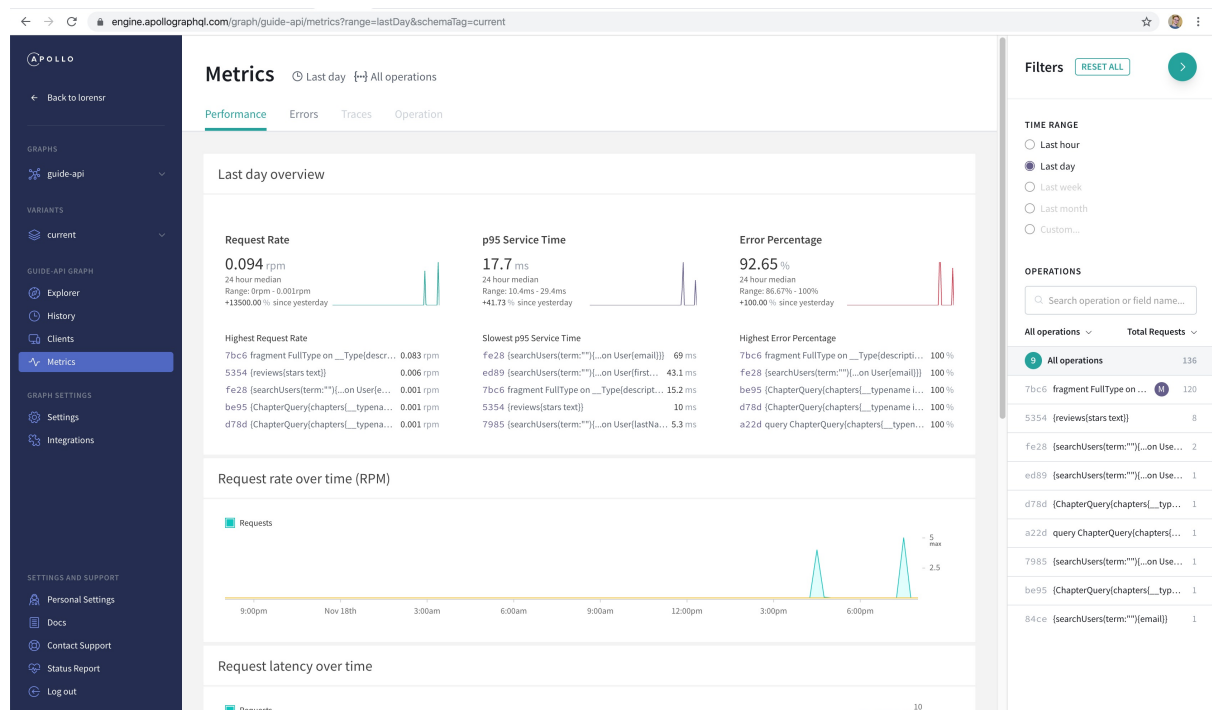
This command sends Apollo our schema, which is used for GraphQL analytics and other Graph Manager features like [schema change validation](#). When we change our schema, we need to re-run the command. Usually this is done automatically as part of [continuous integration](#) ([CircleCI example](#)).

Now we can make queries in Playground, reload [Graph Manager](#), select "Metrics" from the menu, and see server analytics!

If we only want to see production analytics, we can remove `ENGINE_API_KEY` from `.env` and set it on Heroku:

```
$ heroku config:set ENGINE_API_KEY="service:guide-api:*****"
```

Here's an example metrics dashboard:



We see:

- A low total request rate of 0.094 rpm (requests per minute). The operation with the highest request rate (0.083 rpm) begins with `fragment FullType`, and it has 120 total requests, which we can see on the right in the Filters sidebar.
- A low p95 service time of 17.7ms, which means 95% of requests are responded to within 17.7ms.
- A high error rate of 92.65%. Most of the errors come from the `fragment FullType` operation, which is sent by Playground to request the schema (and fails because introspection is disabled on this production server).
- The request rate over time, and after we scroll down, request latency over time and request latency distribution.

We can also see how difficult it is to differentiate unnamed queries—for instance, the four different `searchUsers` queries. To see which query has the second-slowest service time, we'd need to select it and then click on the “Operation” tab:

## Metrics 🕒 Last day {...} ed89 (unspecified operation name)

Performance
Errors
Traces
Operation

### Operation signature

```

{
  searchUsers(term: "") {
    ... on User {
      firstName
    }
  }
}

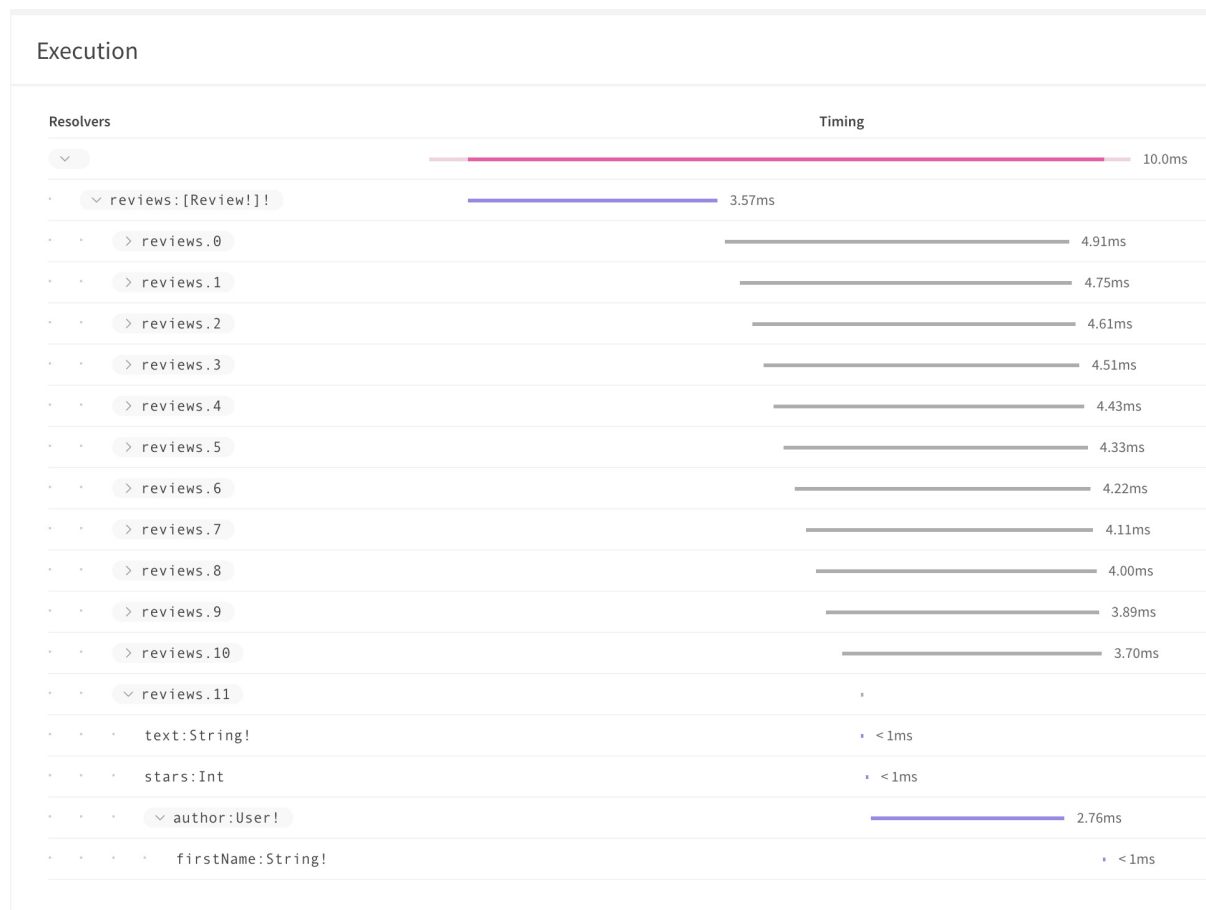
```

The “Traces” tab shows us the timeline of when resolvers are called and how long they take to complete. Here’s a `reviews` query and its trace:

```

{
  reviews {
    text
    stars
    author {
      firstName
    }
  }
}

```



The `reviews` resolver fetches the list of reviews, which takes 3.57ms, and then Apollo Server calls `Review.*` field resolvers, starting with the first review (`reviews.0` in the trace), and ending with `reviews.11`, which is expanded so that we can see the timing of the field resolvers. `Review.text` and `Review.stars` return immediately, since they're just fields on the review object, but `Review.author` requires a database lookup. That lookup is actually done in a single query for all reviews 0–11, as all the reviews have the same author and our datasource uses `Dataloader`, which deduplicates the 12 identical author queries. The query takes 2.76ms, after which the `User.firstName` resolver returns immediately, and the entire query response is ready to send to the client.

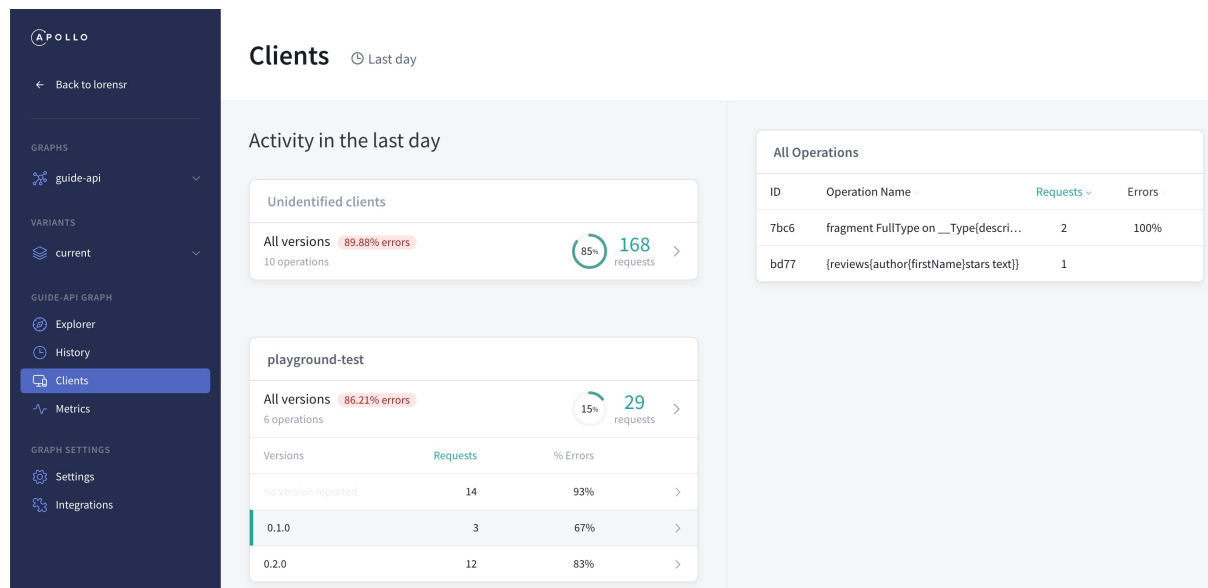
The Filters sidebar lets us filter by time range or by operation, but we can also filter by client type and version. To do that, we select “Clients” from the left sidebar. Now clients are listed on the left half of the page. Currently we only see one labeled “Unidentified clients” and “All versions.” That’s because none of our clients have identified themselves yet. They can do so by setting two headers, `apollographql-client-name` (like “webapp”, “iOS-app”, “marketing-script”, etc.) and `apollographql-client-version` (like `0.1.0`, `v2`, etc.).

Let’s open the HTTP headers section of Playground and enter these:

```
{
  "apollographql-client-name": "playground-test",
  "apollographql-client-version": "0.1.0"
}
```

When using Apollo Client, we can use the `name` and `version` constructor options: `new ApolloClient({ link, cache, name: 'web', version: '1.0' })`.

Then, if we run a query, change the version, run more queries, and refresh Graph Manager, we’ll see the new client type with two versions:



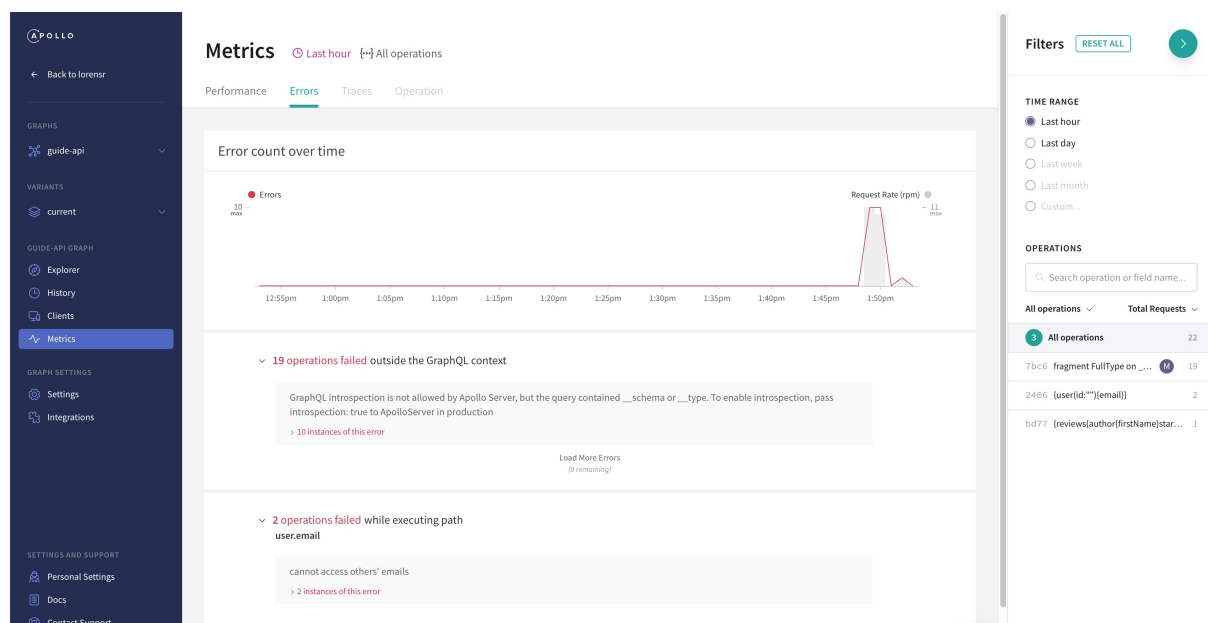
Selecting a version and then an operation on the right takes us to the metrics page of that query for that client version. We can also look at other operations used by that client in the Filters sidebar.

## Error reporting

If you're jumping in here, `git checkout 28_0.1.0` (tag [28\\_0.1.0](#), or compare [28...29](#))

In this section we'll look at what kind of error reporting Graph Manager covers, and then we'll look at a dedicated error reporting service.

In the [last section](#) we set up Apollo's Graph Manager and looked at its analytics. The one tab of the Metrics page we didn't get to is the Errors tab:



The general errors page (without an operation selected) shows a timeline of total error count, followed by a list of all errors within the current time range, grouped by where they occurred—either in a specific resolver, like the `user.email` errors at the bottom, or before the server starts calling resolvers (labeled as “outside of the GraphQL



context” above). The latter category often includes failures parsing or validating the request’s operation. In this example, the validation fails because the operation includes a `__schema` root Query field, but the field is not in the schema because introspection is turned off.

We can expand the instances links to get a list of times and operations in which the error occurred:

▼ **2 operations failed** while executing path  
user.email

cannot access others' emails			
▼ <b>2 instances of this error</b>			
Time	Operation		Trace
2019-11-20 7:06 pm -05:00	2406 {user(id:""){email}}		.0.25d
2019-11-20 7:07 pm -05:00	2406 {user(id:""){email}}		.0.25e

And when we have an operation selected, the Errors tab only shows us errors that occurred during the execution of that operation.

There are a few features that Graph Manager doesn’t have that would be useful:

- Stack traces
- The contents of the `extensions` field of the GraphQL error (above we only see the `message` field)
- The ability to attach further information, like the current user
- The ability to ignore errors or mark them as fixed
- Team features like the ability to attach notes or assign errors to people
- The ability to search through the errors

There are a few error-tracking services that provide these features. We’ll set up [Sentry](#)—one of the most popular ones—but setting up another service would work similarly.

First we [create an account](#), and then we create our first Sentry project, choosing Node.js as the project type. We’re given a statement like `Sentry.init({ dsn: 'https://...' })` with our new project’s ID filled in, which we paste into our code:

`src/formatError.js`

```
import * as Sentry from '@sentry/node'

Sentry.init({
  dsn: 'https://ceb14feec00b4c49bebd10a9674bb68d@sentry.io/5168151'
})
```

Now Sentry automatically gathers uncaught errors like this one:

```
Sentry.init({
  dsn: 'https://ceb14feec00b4c49bebd10a9674bb68d@sentry.io/5168151'
})

myUndefinedFunction()
```

Within seconds of `npm run dev`, we should see a new error in our Sentry dashboard:

The screenshot displays the Sentry web interface for an issue titled "ReferenceError Object.myUndefinedFunction(formatErr...". The error message is "myUndefinedFunction is not defined". The interface shows 1 event, 0 users, and 0 assignees. The stack trace indicates the error occurred in "formatError.js" at line 11. The right sidebar provides additional context, including ownership rules and a timeline of the error occurrence.

We see the time, error message, stack trace, and line of code. And if the same error happens again, it will be grouped with this one so that we can see the total number of occurrences and graph occurrences over time.

This is all really useful, but the issue is that Apollo Server catches all errors that occur during GraphQL requests, which is where most of our errors will occur. Since Sentry is only gathering uncaught errors, it misses most of our errors. To tell Sentry about those errors, we can use one of two `ApolloServer()` options:

- `formatError` function
- `plugins` array with a new plugin we write

The first is simpler, and we're already using it:

`src/index.js`

```
import formatError from './formatError'

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError,
  cache
})
```

`src/formatError.js`

```
export default error => {
  if (inProduction) {
    // send error to tracking service
  } else {
    console.log(error)
    console.log(get(error, 'extensions.exception.stacktrace'))
  }
}
```

```
const name = get(error, 'extensions.exception.name') || ''
if (name.startsWith('Mongo')) {
  return new InternalServerError()
} else {
  return error
}
}
```

We're currently using the `formatError()` function to log errors in development and mask errors involving MongoDB. We can call `Sentry.captureException()` to tell Sentry about errors:

```
import get from 'lodash/get'
import * as Sentry from '@sentry/node'
import { AuthenticationError, ForbiddenError } from 'apollo-server'

import { InternalServerError, InputError } from './util/errors'

const NORMAL_ERRORS = [AuthenticationError, ForbiddenError, InputError]
const NORMAL_CODES = ['GRAPHQL_VALIDATION_FAILED']
const shouldReport = e =>
  !NORMAL_ERRORS.includes(e.originalError) &&
  !NORMAL_CODES.includes(get(e, 'extensions.code'))

export default error => {
  if (inProduction) {
    if (shouldReport(error)) {
      Sentry.captureException(error.originalError)
    }
  } else {
    console.log(error)
    console.log(get(error, 'extensions.exception.stacktrace'))
  }
  ...
}
```

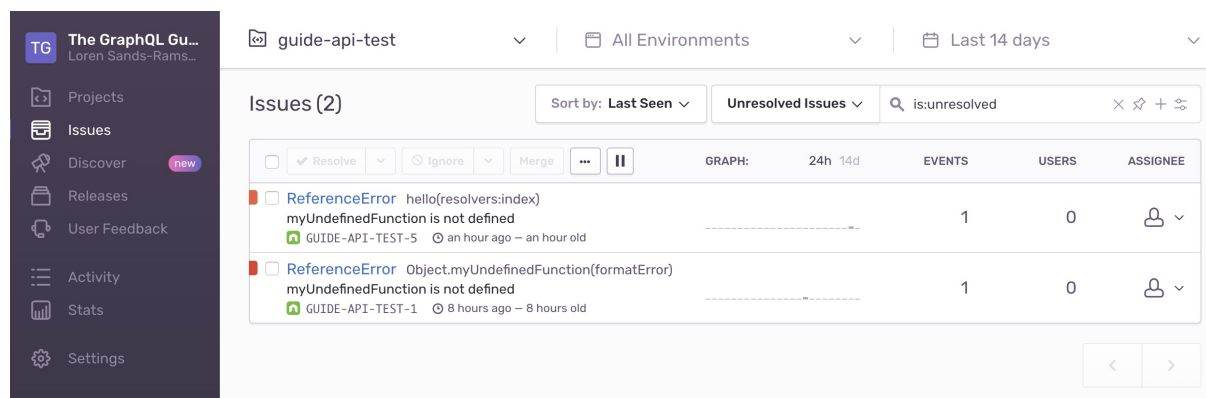
The `error` the function receives is the GraphQL error that's included in the response to the client. To get the Node.js error object (which is what Sentry expects), we do `error.originalError`. We also use `shouldReport()` to avoid reporting normal errors, like auth and query format errors, since we don't need to track and fix them.

If we had a public API, we might want to track query-parsing errors in case we find that developers consistently make certain mistakes, in which case we could try to improve our schema or documentation.

To test, we can run `NODE_ENV=production npm run dev` and add an error to `Query.hello`:

`src/resolvers/index.js`

```
const resolvers = {
  Query: {
    hello: () => ' ' && myUndefinedFunction(),
    isoString: (_, { date }) => date.toISOString()
  }
}
```



We can see the error message is the same, but the new entry shows a different function and file:

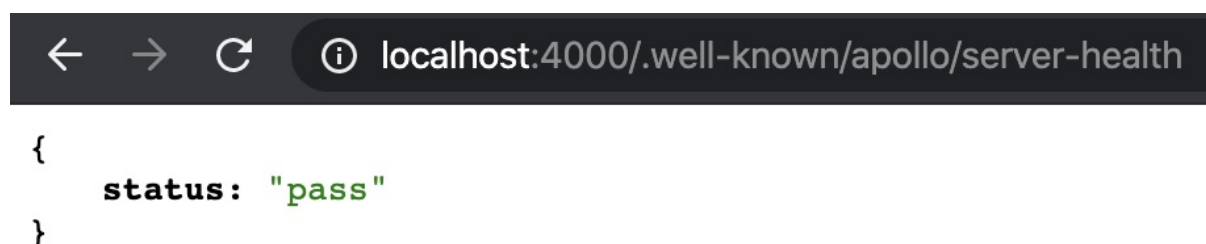
```
hello(resolvers:index) .
```

If we want to track more information in Sentry, like details about the request and context (such as the current user), then we need to use a plugin instead of `formatError`. We use the `plugins` option:

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
  context,
  formatError,
  cache,
  plugins: [sentryPlugin]
})
```

And we create `sentryPlugin` according to the [plugin docs](#), defining the `didEncounterErrors()` method and using `Sentry.withScope()`.

One last thing to consider is that if our server is not running—if something happened to our Node.js process or our machine—we won't receive errors in Sentry. In many cases we won't need to worry about this: for instance, a Node.js PaaS will automatically monitor and restart the process, and for a FaaS, it's irrelevant. But if it is relevant for our deployment setup, we can use an uptime / monitoring service that pings our server to see if it's still reachable over the internet and responsive. The URL we can use for that (as well as for a load balancer, if we're using one) is `/.well-known/apollo/server-health`, which should return status 200 and this JSON:



## More data sources

- [SQL](#)
  - [SQL setup](#)
  - [SQL data source](#)
  - [SQL testing](#)
  - [SQL performance](#)
- [REST](#)

- [GraphQL](#)
- [Custom data source](#)
- [Prisma](#)

There are lots of other sources of data out there we might want to use in our GraphQL servers, and when we want to query one, we use a *data source*. When we use the term “data source” in this chapter, we’re usually talking about a JavaScript class that has Apollo’s `DataSource` class as an ancestor, like the `MongoDataSource` we [used earlier](#). There are data sources on npm that others have written, and we can write our own. There are also alternatives, one of which we’ll cover at the end called Prisma.

## SQL

Background: [SQL](#)

Contents:

- [SQL setup](#)
- [SQL data source](#)
- [SQL testing](#)
- [SQL performance](#)

In this section we replace our use of MongoDB with SQL. In the first part we’ll get our SQL database and table schemas set up. Then we’ll replace our use of `MongoDataSource` with `SQLDataSource`. Then in [SQL testing](#), we update our tests, and, finally in [SQL performance](#), we improve our server’s database querying.

### SQL setup

If you’re jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...sql](#))

A [SQL database](#) takes more setup than the MongoDB database we’ve been using: We need to write *migrations*—code that creates or alters tables and their schemas. The most popular Node library for SQL is [Knex](#), and it includes the ability to write and run migrations. To start using it, we run `knex init`. Since we already have it in our `node_modules/`, we can run `npx knex init` in a new directory within our repository:

```
$ mkdir sql
$ cd sql/
$ npx knex init
```

This creates a config file:

`sql/knexfile.js`

```
// Update with your config settings.

module.exports = {
  development: {
    client: 'sqlite3',
    connection: {
      filename: './dev.sqlite3'
    },
  },

  staging: {
    client: 'postgresql',
    connection: {
      database: 'my_db',
      user: 'username',
      password: 'password'
    },
    pool: {
```

```

    min: 2,
    max: 10
  },
  migrations: {
    tableName: 'knex_migrations'
  }
},

production: {
  client: 'postgresql',
  ...
}
}

```

By default, it uses **SQLite** and **PostgreSQL** (two types of SQL databases) for development and deployment, respectively.

One aspect of database setup that's easier with SQL than MongoDB is running the database in development—SQLite doesn't need to be installed with Homebrew and run as a service. Instead, it can be installed with a Node library and run off of a single file. So unless we're using a special feature that PostgreSQL supports but SQLite doesn't, we can use SQLite in development.

We also won't be deploying, so all we need is:

`sql/knexfile.js`

```

module.exports = {
  development: {
    client: 'sqlite3',
    connection: {
      filename: './dev.sqlite3'
    },
    useNullAsDefault: true
  }
}

```

(We added `useNullAsDefault: true` to avoid a warning message.)

Now we can use Knex to create a migration that will set up our users and reviews tables:

```
$ npx knex migrate:make users_and_reviews
```

This generates a file in the following format:

`sql/migrations/[timestamp]_users_and_reviews.js`

```

exports.up = function(knex) {
}

exports.down = function(knex) {
}

```

Inside the `up` function, we create the two tables, and inside the `down` function, we *drop* (delete) them. To do all that, we use Knex's [schema-building API](#):

`sql/migrations/20191228233250_users_and_reviews.js`

```

exports.up = async knex => {
  await knex.schema.createTable('users', table => {
    table.increments('id')
    table.string('first_name').nullable()
    table.string('last_name').nullable()
  })
}

```

```

    table.string('username').notNullable()
    table.string('email')
    table
      .string('auth_id')
      .notNullable()
      .unique()
    table.datetime('suspended_at')
    table.datetime('deleted_at')
    table.integer('duration_in_days')
    table.timestamps()
  })
}

```

- `knex.schema.createTable('users')` creates a table named `users`.
- `table.increments('id')` creates a primary index column named `id`. It's auto-incrementing, which means the first record that's inserted is given an `id` of 1, and the second record gets an `id` of 2, etc.
- `table.string('first_name').notNullable()` creates a `first_name` column that can hold a string and can't be null.
- `table.string('auth_id').notNullable().unique()` creates an `auth_id` non-nullable string column that has to be unique among all records in the table.
- `table.datetime('suspended_at')` creates a `suspended_at` column that can hold a datetime.
- `table.timestamps()` creates `created_at` and `updated_at` datetime columns.

Similarly, we can create the `reviews` table:

```

exports.up = async knex => {
  await knex.schema.createTable('users', table => { ... })
  await knex.schema.createTable('reviews', table => {
    table.increments('id')
    table
      .integer('author_id')
      .unsigned()
      .notNullable()
      .references('id')
      .inTable('users')
    table.string('text').notNullable()
    table.integer('stars').unsigned()
    table.timestamps()
  })
}

```

The below part sets up a *foreign key constraint* on `author_id`, so the only values that can be stored in this column match an `id` field in the `users` table:

```

table
  .integer('author_id')
  .unsigned()
  .notNullable()
  .references('id')
  .inTable('users')

```

Finally, we call `dropTable()` in the `down` function:

[sql/migrations/20191228233250\\_users\\_and\\_reviews.js](#)

```

exports.up = async knex => {
  await knex.schema.createTable('users', table => { ... })
  await knex.schema.createTable('reviews', table => { ... })
}

exports.down = async knex => {
  await knex.schema.dropTable('users')
  await knex.schema.dropTable('reviews')
}

```

To run our migration `up` function, we use:

```
$ npx knex migrate:latest
```

And to undo, we would do `npx knex migrate:rollback --all`. If in the future we want to make a change to the schema, we would create another migration with a more recent timestamp—e.g.,

`[timestamp]_add_deleted_column_to_reviews.js`—that adds a `deleted` column to the `reviews` table, and commits it to git. Then, whenever a dev was on the version of the code that used the `reviews.deleted` column, they could migrate to the latest version of the database, and code that modifies a review's `deleted` field would work.

With MongoDB, we didn't have migrations, and we added or changed documents manually. With SQL, we could run migrations that drop our tables and everything in them, so re-inserting records manually would get tedious. So Knex supports *seed files* that we can run to automatically insert records. We start with `seed:make`, which creates an example seed file:

```
$ npx knex seed:make users
```

`sql/seeds/users.js`

```
exports.seed = function(knex) {
  // Deletes ALL existing entries
  return knex('table_name').del()
    .then(function () {
      // Inserts seed entries
      return knex('table_name').insert([
        {id: 1, colName: 'rowValue1'},
        {id: 2, colName: 'rowValue2'},
        {id: 3, colName: 'rowValue3'}
      ]);
    });
};
```

Now we modify the example file to use `async/await` and match our `users` table schema:

```
exports.seed = async knex => {
  await knex('users').del()
  await knex('users').insert([
    {
      id: 1,
      firstName: 'John',
      lastName: 'Resig',
      username: 'jeresig',
      email: 'john@graphql.guide',
      authId: 'github|1615',
      created_at: new Date(),
      updated_at: new Date()
    }
  ])
}
```

And then copy the file for inserting reviews:

`sql/seeds/reviews.js`

```
exports.seed = async knex => {
  await knex('reviews').del()
  await knex('reviews').insert([
    {
      id: 1,
      author_id: 1,
      text: 'Now that's a downtown job!',
      stars: 5,
      created_at: new Date(),
    }
  ])
}
```



```

    updated_at: new Date()
  },
  {
    id: 2,
    author_id: 1,
    text: 'Passable',
    stars: 3,
    created_at: new Date(),
    updated_at: new Date()
  }
])
}

```

We run the seed files with:

```
$ npx knex seed:run
```

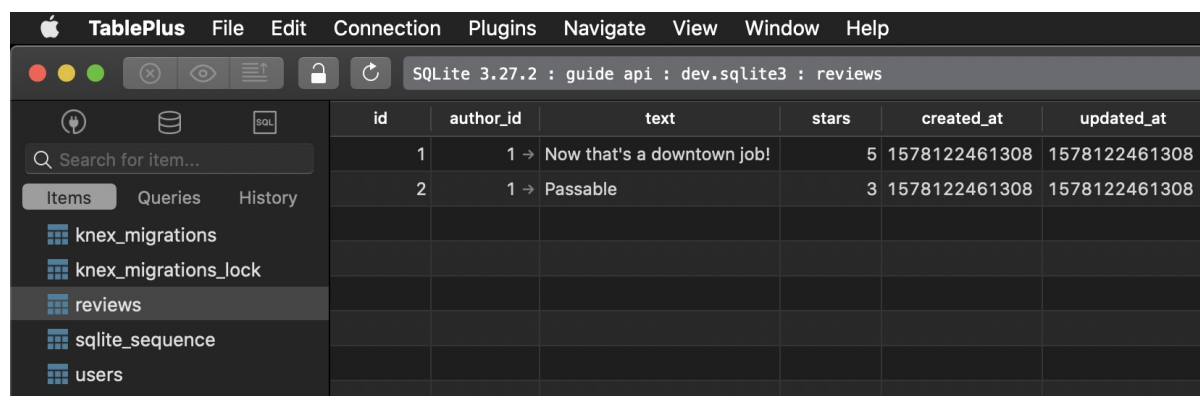
We can view if it worked with either the command-line SQLite client or a GUI. The command-line client, `sqlite3`, is included by default on Macs. We give it the database file `sql/dev.sqlite3` as an argument, and then we can run SQL statements like `SELECT * FROM reviews;`

```

$ sqlite3 sql/dev.sqlite3
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
sqlite> SELECT * FROM reviews;
1|1|Now that's a downtown job!|5|1578122461308|1578122461308
2|1|Passable|3|1578122461308|1578122461308

```

There are many SQL GUIs. Our favorite is [TablePlus](#), which works with not only different types of SQL databases, but other databases as well, including Redis and MongoDB. When creating a new connection, we select SQLite and then the file `sql/dev.sqlite3`, and hit Connect. Then on the left, we see the list of tables in our database, and if we double-click `reviews`, we see the table's contents:



id	author_id	text	stars	created_at	updated_at
1	1 →	Now that's a downtown job!	5	1578122461308	1578122461308
2	1 →	Passable	3	1578122461308	1578122461308

Lastly, we no longer need to connect to a MongoDB database, so we can remove the call to `connectToDB()` in `src/index.js`.

Before we commit our changes, we want to add the below line to `.gitignore`:

```
sql/dev.sqlite3
```

We don't want our database in our code repository—it's meant to be generated and modified by each individual developer using our migration and seed scripts.

## SQL data source

If you're jumping in here, `git checkout sql_0.1.0` (tag `sql_0.1.0`, or compare `sql...sql2`)

Now that we've set up our SQL database and inserted records, we need to query them. So we look for a SQL data source class to use, either on the [community data sources list](#) in the Apollo docs or by searching “apollo data source sql” on Google or npm. We find `datasource-sql`, which provides the class `SQLDataSource`.

`SQLDataSource` is unusual among data sources in that:

- A single instance is created (versus a new instance for each request).
- It does caching only, not batching.

It also:

- recommends using a single class for the whole database, instead of a class per table as we did with `MongoDataSource`
- uses a specific library—Knex!

Let's start by creating our data source class:

`src/data-sources/SQL.js`

```
import { SQLDataSource } from 'datasource-sql'

class SQL extends SQLDataSource {
  // TODO
}

export default SQL
```

Our job will be to fill in the class with methods our resolvers need. To know what those methods are, let's go at it from the other direction: creating and using the data source as if it were complete.

First let's create it. Instead of our current data sources file:

`src/data-sources/index.js`

```
import Reviews from './Reviews'
import Users from './Users'
import Github from './Github'
import { db } from '../db'

export default () => ({
  reviews: new Reviews(db.collection('reviews')),
  users: new Users(db.collection('users'))
})

export { Reviews, Users, Github }
```

we do:

```
import Github from './Github'
import SQL from './SQL'

export const knexConfig = {
  client: 'sqlite3',
  connection: {
    filename: './sql/dev.sqlite3'
  },
  useNullAsDefault: true
}

export const db = new SQL(knexConfig)

export default () => ({ db })

export { Github }
```

The `SQLDataSource` constructor takes the same config we have in our `sql/knexfile.js`. Since we only want a single instance, we move the creation (`new SQL(knexConfig)`) outside of the exported function. Instead of the data source instances being named `reviews` and `users`, it's named `db` (because it is the way to access the whole SQL database).

Now in resolvers, we can use functions like `context.dataSources.db.getReviews()` instead of

`context.dataSources.reviews.all()`. And we also need to replace `camelCase` fields with `snake_case`, like `deletedAt` -> `deleted_at`.

`src/resolvers/User.js`

```
export default {
  Query: {
    me: ...,
    user: (_, { id }, { dataSources: { db } }) => db.getUser({ id }),
    searchUsers: (_, { term }, { dataSources: { db } }) => db.searchUsers(term)
  },
  UserResult: {
    __resolveType: result => {
      if (result.deleted_at) {
        return 'DeletedUser'
      } else if (result.suspended_at) {
        return 'SuspendedUser'
      } else {
        return 'User'
      }
    }
  },
  SuspendedUser: {
    daysLeft: user => {
      const end = addDays(user.suspended_at, user.duration_in_days)
      return differenceInDays(end, new Date())
    }
  },
  User: {
    firstName: user => user.first_name,
    lastName: user => user.last_name,
    email: ...,
    photo(user) {
      // user.auth_id: 'github|1615'
      const githubId = user.auth_id.split('|')[1]
      return `https://avatars.githubusercontent.com/u/${githubId}`
    },
    createdAt: user => user.created_at,
    updatedAt: user => user.updated_at
  },
  Mutation: {
    createUser(_, { user, secretKey }, { dataSources: { db } }) {
      if (secretKey !== process.env.SECRET_KEY) {
        throw new AuthenticationError('wrong secretKey')
      }

      return db.createUser(user)
    }
  }
}
```

So the `db.*` methods we needed and named are:

```
db.getUser()
db.searchUsers()
db.createUser()
```

Note that we needed to add resolvers for `firstName`, `lastName`, and `updatedAt`, because we no longer have database fields with those exact names (instead we have `first_name`, `last_name`, and `updated_at`).

Next let's update our Review resolvers:

src/resolvers/Review.js

```

export default {
  Query: {
    reviews: (_, __, { dataSources: { db } }) => db.getReviews()
  },
  Review: {
    author: (review, _, { dataSources: { db } }) =>
      db.getUser({ id: review.author_id }),
    fullReview: async (review, _, { dataSources: { db } }) => {
      const author = await db.getUser({ id: review.author_id })
      return `${author.first_name} ${author.last_name} gave ${review.stars} stars, saying: "${review.text}"`
    },
    createdAt: review => review.created_at,
    updatedAt: review => review.updated_at
  },
  Mutation: {
    createReview: (_, { review }, { dataSources: { db }, user }) => {
      ...

      const newReview = db.createReview(review)

      pubsub.publish('reviewCreated', {
        reviewCreated: newReview
      })

      return newReview
    }
  },
  Subscription: {
    reviewCreated: ...
  }
}

```

We reused the `db.getUser()` function and used two new ones:

```

db.getReviews()
db.createReview()

```

The Users and Reviews resolvers were the only place we used `context.dataSources`, but we can do a workspace text search for `db.collection` to find any other uses of our MongoDB database. The only match is from our context function in `src/context.js`:

```

const user = await db.collection('users').findOne({ authId })

```

To update this, we need access to our SQL data source. In `src/data-sources/index.js`, we have this line:

```

export const db = new SQL(knexConfig)

```

So we can import our new `db` from there.

src/context.js

```

import { db } from '../data-sources/'

export default async ({ req }) => {
  ...

  const user = await db.getUser({ auth_id: authId })

  ...

  return context
}

```

Now we can implement all the data source methods we're using:

```
db.getReviews()
db.createReview()
db.createUser()
db.getUser()
db.searchUsers()
```

Inside methods we have access to `this.context`, which has the current user, and `this.knex`, our [Knex instance](#), which we use to construct SQL statements. For example, here's `SELECT * FROM reviews;`:

`src/data-sources/SQL.js`

```
import { SQLDataSource } from 'datasource-sql'

const REVIEW_TTL = 60 // minute

class SQL extends SQLDataSource {
  getReviews() {
    return this.knex
      .select('*')
      .from('reviews')
      .cache(REVIEW_TTL)
  }

  async createReview(review) { ... }
  async createUser(user) { ... }
  async getUser(where) { ... }
  searchUsers(term) { ... }
}

export default SQL
```

The added `.cache()` tells `SQLDataSource` to cache the response for the provided number of seconds.

Next up is `createReview()`, where we get a review from the client and need to add the current user's ID as well as timestamps:

```
class SQL extends SQLDataSource {
  getReviews() { ... }

  async createReview(review) {
    review.author_id = this.context.user.id
    review.created_at = Date.now()
    review.updated_at = Date.now()
    const [id] = await this.knex
      .returning('id')
      .insert(review)
      .into('reviews')
    review.id = id
    return review
  }

  async createUser(user) { ... }
  async getUser(where) { ... }
  searchUsers(term) { ... }
}
```

We tell Knex to return the inserted ID ( `.returning('id')` ) so that we can add it to the review object and return it. We didn't do this before because MongoDB's `collection.insertOne(review)` automatically added an `_id` to `review`. We do the same for `createUser()`:

```
class SQL extends SQLDataSource {
  getReviews() { ... }
  async createReview() { ... }
```

```

async createUser(user) {
  const newUser = {
    first_name: user.firstName,
    last_name: user.lastName,
    username: user.username,
    email: user.email,
    auth_id: user.authId,
    created_at: Date.now(),
    updated_at: Date.now()
  }

  const [id] = await this.knex
    .returning('id')
    .insert(newUser)
    .into('users')
  newUser.id = id

  return newUser
}

async getUser(where) { ... }
searchUsers(term) { ... }
}

```

Here we just take the fields out of the `user` argument (which matches the GraphQL schema) and put them into a `newUser` object that matches the SQL `users` table schema.

Lastly, we have `getUser()` and `searchUser()`. The `getUser()` function receives an object like `{id: 1}` or `{auth_id: 'github|1615'}`, which can be passed directly to Knex's `.where()`:

```

const REVIEW_TTL = 60 // minute
const USER_TTL = 60 * 60 // hour

class SQL extends SQLDataSource {
  getReviews() { ... }
  async createReview(review) { ... }
  async createUser(user) { ... }

  async getUser(where) {
    const [user] = await this.knex
      .select('*')
      .from('users')
      .where(where)
      .cache(USER_TTL)

    return user
  }

  searchUsers(term) {
    return this.knex
      .select('*')
      .from('users')
      .where('first_name', 'like', `>${term}%`)
      .orWhere('last_name', 'like', `>${term}%`)
      .orWhere('username', 'like', `>${term}%`)
      .cache(USER_TTL)
  }
}

```

We use a longer TTL for users with the idea they'll change less often than reviews will. We could also have different TTLs for different types of queries. For instance, we could use 60 seconds for selecting a single review but only 5 seconds for selecting all reviews. Then we wouldn't have to wait more than 5 seconds to see a new review appear on the reviews page.

SQL's `like` syntax is followed by a search pattern that can include the `%` wildcard, which takes the place of zero or more characters.

Now let's see if it works by running `npm run dev` and making queries in Playground:

```

1 {
2   reviews {
3     id
4     text
5     stars
6     author {
7       firstName
8     }
9   }
10 }

```

```

{
  "data": {
    "reviews": [
      {
        "id": "1",
        "text": "Now that's a downtown job!",
        "stars": 5,
        "author": {
          "firstName": "John"
        }
      },
      {
        "id": "2",
        "text": "Passable",
        "stars": 3,
        "author": {
          "firstName": "John"
        }
      }
    ]
  }
}

```

😊 Looks like it's working! ...but not if we select a Date field:

```

1 {
2   reviews {
3     createdAt
4   }
5 }

```

```

{
  "errors": [
    {
      "message": "Resolvers for Date scalars must return JavaScript Date objects",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ],
      "path": [
        "reviews",
        0,
        "createdAt"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "exception": {
          "stacktrace": [
            "Error: Resolvers for Date scalars must return JavaScript Date objects",
          ]
        }
      }
    }
  ]
}

```

😞 The stacktrace points to this part of `src/resolvers/Date.js` :

```

serialize(date) {
  if (!(date instanceof Date)) {
    throw new Error(
      'Resolvers for Date scalars must return JavaScript Date objects'
    )
  }

  if (!isValid(date)) {
    throw new Error('Invalid Date scalar')
  }

  return date.getTime()
}

```

Remember when we [wrote that](#)? A custom scalar's `serialize()` function is called when a value is returned from a resolver, and it formats the value for being sent to the client. When we were querying MongoDB, our results—for instance `review.createdAt`—were JavaScript Date objects, and we formatted them as integers. But when we query SQL datetime fields, we get them as integers, so we don't need to format them differently for sending to the client. Similarly, when we receive values from the client, we don't need to convert them to Date objects in `parseValue()` and `parseLiteral()`. However, we can still check to make sure they're valid date integers:

`src/resolvers.Date.js`

```
import { GraphQLScalarType } from 'graphql'
import { Kind } from 'graphql/language'

const isValid = date => !isNaN(date.getTime())

export default {
  Date: new GraphQLScalarType({
    name: 'Date',
    description:
      'The `Date` scalar type represents a single moment in time. It is serialized as an integer, equal to the number of milliseconds since the Unix epoch.',

    parseValue(value) {
      if (!Number.isInteger(value)) {
        throw new Error('Date values must be integers')
      }

      const date = new Date(value)
      if (!isValid(date)) {
        throw new Error('Invalid Date value')
      }

      return value
    },

    parseLiteral(ast) {
      if (ast.kind !== Kind.INT) {
        throw new Error('Date literals must be integers')
      }

      const dateInt = parseInt(ast.value)
      const date = new Date(dateInt)
      if (!isValid(date)) {
        throw new Error('Invalid Date literal')
      }

      return dateInt
    },

    serialize(date) {
      if (!Number.isInteger(date)) {
        throw new Error('Resolvers for Date scalars must return integers')
      }

      if (!isValid(new Date(date))) {
        throw new Error('Invalid Date scalar')
      }

      return date
    }
  })
}
```

For `parseValue()`, the value is already an integer. For `parseLiteral()`, we get a string, so we use `parseInt()`.

The last thing we need to update is our root query field `isoString(date: Date)`:

```
isoString: (_, { date }) => date.toISOString()
```



`date` used to be a `Date` instance, but now it's an integer, so we can't call `toISOString()` until we create a `Date` object. But strangely enough, we can't create a `Date` object because the `date` identifier is being used later in the file:

```
import Date from './Date'
```

So we also need to change what we call the `Date` resolvers we're importing:

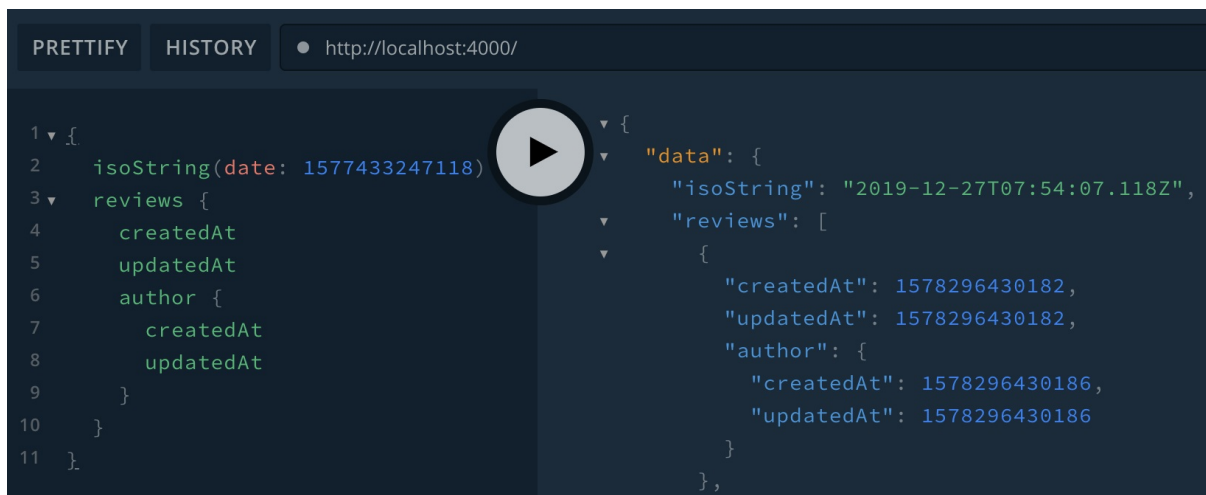
`src/resolvers/index.js`

```
const resolvers = {
  Query: {
    hello: () => ' ',
    isoString: (_, { date }) => new Date(date).toISOString()
  }
}

import Review from './Review'
import User from './User'
import DateResolvers from './Date'
import Github from './Github'

export default [resolvers, Review, User, DateResolvers, Github]
```

Now all our dates are working:



## SQL testing

If you're jumping in here, `git checkout sql2_0.1.0` (tag `sql2_0.1.0`, or compare `sql2...sql3`)

In the last section we implemented and used (okay, more like used then implemented 😊) our SQL data source. We also made a couple of queries to see if it worked, and the queries did work (eventually), but it wasn't a comprehensive test. Let's update our automated tests (which are currently broken) so we can have a higher level of confidence in our code's correctness.

The place to start updating is in the code at the base of all our tests, `test/guide-test-utils.js`. We need to:

- Update mocked data field names (`_id` -> `id` and `firstName` -> `first_name`) and values.
- Mock our new SQL data source.
- Remove our old data sources and database connection code.

`test/guide-test-utils.js`

```
import { ApolloServer } from 'apollo-server'
import { promisify } from 'util'
```

```

import { HttpLink } from 'apollo-link-http'
import fetch from 'node-fetch'
import { execute, toPromise } from 'apollo-link'

import {
  server,
  typeDefs,
  resolvers,
  context as defaultContext,
  formatError
} from '../src/'

const created_at = new Date('2020-01-01').getTime()
const updated_at = created_at

export const mockUser = {
  id: 1,
  first_name: 'First',
  last_name: 'Last',
  username: 'mockA',
  auth_id: 'mockA|1',
  email: 'mockA@gmail.com',
  created_at,
  updated_at
}

const mockUsers = [mockUser]

const reviewA = {
  id: 1,
  text: 'A+',
  stars: 5,
  created_at,
  updated_at,
  author_id: mockUser.id
}

const reviewB = {
  id: 2,
  text: 'Passable',
  stars: 3,
  created_at,
  updated_at,
  author_id: mockUser._id
}

const mockReviews = [reviewA, reviewB]

class SQL {
  getReviews() {
    return mockReviews
  }
  createReview() {
    return reviewA
  }
  createUser() {
    return mockUser
  }
  getUser() {
    return mockUser
  }
  searchUsers() {
    return mockUsers
  }
}

export const db = new SQL()

export const createTestServer = ({ context = defaultContext } = {}) => {
  const server = new ApolloServer({
    typeDefs,
    resolvers,
    dataSources: () => ({ db }),
    context,
  })

```

```

    formatError,
    engine: false
  })

  return { server, dataSources: { db } }
}

export const startE2EServer = async () => {
  const e2eServer = await server.listen({ port: 0 })

  const stopServer = promisify(e2eServer.server.close.bind(e2eServer.server))

  const link = new HttpLink({
    uri: e2eServer.url,
    fetch
  })

  return {
    stop: stopServer,
    request: operation => toPromise(execute(link, operation))
  }
}

export { createTestClient } from 'apollo-server-testing'
export { default as gql } from 'graphql-tag'

```

In our User resolver tests, we also need to update field names:

`src/resolvers/User.test.js`

```

import {
  createTestServer,
  createTestClient,
  gql,
  mockUser
} from 'guide-test-utils'

const ME = gql`
  query {
    me {
      id
    }
  }
`

test('me', async () => {
  const { server } = createTestServer({
    context: () => ({ user: { id: 'itme' } })
  })
  const { query } = createTestClient(server)

  const result = await query({ query: ME })
  expect(result.data.me.id).toEqual('itme')
})

const USER = gql`
  query User($id: ID!) {
    user(id: $id) {
      id
    }
  }
`

test('user', async () => {
  const { server } = createTestServer()
  const { query } = createTestClient(server)

  const id = mockUser.id
  const result = await query({
    query: USER,
    variables: { id }
  })
  expect(result.data.user.id).toEqual(id.toString())
})

```

```

}))

const CREATE_USER = gql`
  mutation CreateUser($user: CreateUserInput!, $secretKey: String!) {
    createUser(user: $user, secretKey: $secretKey) {
      id
    }
  }
`

test('createUser', async () => {
  const { server } = createTestServer()
  const { mutate } = createTestClient(server)

  const user = {
    firstName: mockUser.first_name,
    lastName: mockUser.last_name,
    username: mockUser.username,
    email: mockUser.email,
    authId: mockUser.auth_id
  }

  const result = await mutate({
    mutation: CREATE_USER,
    variables: {
      user,
      secretKey: process.env.SECRET_KEY
    }
  })

  expect(result).toMatchSnapshot()
})

```

Now if we run `npm test`, we see tests fail due to mismatching snapshots, which we can update with `npx jest -u`.

One thing we updated in the last section that we don't have a test for is the context function:

`src/context.js`

```

import { AuthenticationError } from 'apollo-server'

import { getAuthIdFromJWT } from '../util/auth'
import { db } from '../data-sources/'

export default async ({ req }) => {
  const context = {}

  const jwt = req && req.headers.authorization
  let authId

  if (jwt) {
    try {
      authId = await getAuthIdFromJWT(jwt)
    } catch (e) {
      let message
      if (e.message.includes('jwt expired')) {
        message = 'jwt expired'
      } else {
        message = 'malformed jwt in authorization header'
      }
      throw new AuthenticationError(message)
    }
  }

  const user = await db.getUser({ auth_id: authId })
  if (user) {
    context.user = user
  } else {
    throw new AuthenticationError('no such user')
  }
}

return context

```

```
}
```

Let's write a test for it! In order to test it, we have two options:

- Using an authorization header that successfully decodes to our mock `auth_id: mockA|1`. We can't create such a JWT, and, even if we could, it would expire. And then our test would fail.
- Make it a unit test and mock all the functions it calls—in this case `getAuthIdFromJWT()` and `db.getUser()`.

Let's do the second. To mock an import, we need to call `jest.mock(file)`:

```
src/context.test.js
```

```
import { mockUser } from 'guide-test-utils'

jest.mock('./util/auth', () => ({
  getAuthIdFromJWT: jest.fn((jwt) => (jwt === 'valid' ? mockUser.auth_id : null))
}))

jest.mock('./data-sources/', () => ({
  db: {
    getUser: ({ auth_id }) => (auth_id === mockUser.auth_id ? mockUser : null)
  }
}))
```

Now when any code we're testing does the below imports, it will get our mock implementations.

```
import { getAuthIdFromJWT } from './util/auth'
import { db } from './data-sources/'
```

Let's test the success case first:

```
import getContext from './context'
import { getAuthIdFromJWT } from './util/auth'

describe('context', () => {
  it('finds a user given a valid jwt', async () => {
    const context = await getContext({
      req: { headers: { authorization: 'valid' } }
    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
    expect(context.user).toMatchSnapshot()
  })
})
```

We can check our snapshot:

```
src/__snapshots__/context.test.js.snap
```

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`context finds a user given a valid jwt 1`] = `
Object {
  "auth_id": "mockA|1",
  "created_at": 1577836800000,
  "email": "mockA@gmail.com",
  "first_name": "First",
  "id": 1,
  "last_name": "Last",
  "updated_at": 1577836800000,
  "username": "mockA",
}
`;
```

✔ Looks good! Next let's make sure that giving an invalid JWT throws an error:

src/context.test.js

```
import { AuthenticationError } from 'apollo-server'
describe('context', () => {
  it('finds a user given a valid jwt', async () => { ... })

  it('throws error on invalid jwt', async () => {
    const promise = getContext({
      req: { headers: { authorization: 'invalid' } }
    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
    expect(promise).rejects.toThrow(AuthenticationError)
  })
})
```

We see with `npx jest context` that the test fails, saying that the `getAuthIdFromJWT` mock was called twice.

Adding `context` after `npx jest` limits testing to files with “context” in their names.

```
FAIL src/context.test.js
context
  ✓ finds a user given a valid jwt (4ms)
  ✗ throws error on invalid jwt (2ms)

● context > throws error on invalid jwt

expect(received).toBe(expected) // Object.is equality

Expected: 1
Received: 2

   34 |     })
   35 |
>  36 |     expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
      |                                             ^
   37 |     expect(promise).rejects.toThrow(AuthenticationError)
   38 |   })
   39 |

at Object.toBe (src/context.test.js:36:48)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:  1 passed, 1 total
Time:        2.413s
```

The mock calls are cumulative until we clear the mock. Let's do that after each test:

```
describe('context', () => {
  afterEach(() => {
    getAuthIdFromJWT.mockClear()
  })

  it('finds a user given a valid jwt', async () => { ... })

  it('throws error on invalid jwt', async () => {
    const promise = getContext({
      req: { headers: { authorization: 'invalid' } }
    })
```

```

    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
    expect(promise).rejects.toThrow(AuthenticationError)
  })
})

```

✅ And we're back to green. Lastly, let's test a blank auth header:

```

describe('context', () => {
  afterEach(() => {
    getAuthIdFromJWT.mockClear()
  })

  it('finds a user given a valid jwt', async () => { ... })
  it('throws error on invalid jwt', async () => { ... })

  it('is empty without jwt', async () => {
    const context = await getContext({
      req: { headers: {} }
    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(0)
    expect(context).toEqual({})
  })
})

```

✅ And still green! 🐛 All together, that's:

`src/context.test.js`

```

import { AuthenticationError } from 'apollo-server'
import { mockUser } from 'guide-test-utils'

import getContext from './context'
import { getAuthIdFromJWT } from './util/auth'

jest.mock('./util/auth', () => ({
  getAuthIdFromJWT: jest.fn(jwt => (jwt === 'valid' ? mockUser.auth_id : null))
}))

jest.mock('./data-sources/', () => ({
  db: {
    getUser: ({ auth_id }) => (auth_id === mockUser.auth_id ? mockUser : null)
  }
}))

describe('context', () => {
  afterEach(() => {
    getAuthIdFromJWT.mockClear()
  })

  it('finds a user given a valid jwt', async () => {
    const context = await getContext({
      req: { headers: { authorization: 'valid' } }
    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
    expect(context.user).toMatchSnapshot()
  })

  it('throws error on invalid jwt', async () => {
    const promise = getContext({
      req: { headers: { authorization: 'invalid' } }
    })

    expect(getAuthIdFromJWT.mock.calls.length).toBe(1)
    expect(promise).rejects.toThrow(AuthenticationError)
  })

  it('is empty without jwt', async () => {

```

```
const context = await getContext({
  req: { headers: {} }
})

expect(getAuthIdFromJWT.mock.calls.length).toBe(0)
expect(context).toEqual({})
})
})
```

Unfortunately if we run `npm test`, we see our coverage is down to 40%. And if we look at the coverage report ( `npm run open-coverage` ), we see not much of our SQL data source is covered:

**All files src/data-sources**

23.33% Statements 7/30    0% Branches 0/6    10% Functions 1/10    23.33% Lines 7/30

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements
Github.js	18.18%
SQL.js	12.5%
index.js	100%

Our old `Users.js` and `Reviews.js` files were 100% covered:

**All files src/data-sources**

63.64% Statements 14/22    0% Branches 0/6    66.67% Functions 6/9    63.64% Lines 14/22

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements
Github.js	20%
Reviews.js	100%
Users.js	100%
index.js	100%

The issue is that before, we were mocking the `.find()` and `.insertOne()` methods of MongoDB collections, and currently, we're mocking the data source methods:

```
test/guide-test-utils.js

class SQL {
  getReviews() {
    return mockReviews
  }
  createReview() {
    return reviewA
  }
  createUser() {
    return mockUser
  }
}
```



```

    }
    getUser() {
      return mockUser
    }
    searchUsers() {
      return mockUsers
    }
  }
}

```

If we wanted to cover `sql.js`, we would need to run the actual methods, which means we would need to instead mock the `this.knex` used by the methods.

## SQL performance

If you're jumping in here, `git checkout sql3_0.1.0` (tag `sql3_0.1.0`, or compare `sql3...sql4`)

The two main performance factors when it comes to database querying are latency and load. Latency is how quickly we get all the data we need, and load is how much work the database is doing. Latency usually won't be an issue unless we have complex queries or a lot of data. Load won't be an issue unless we have a lot of clients simultaneously using our API.

When neither latency nor load is an issue for our app, we don't need to concern ourselves with performance, and our current implementation is fine. If either becomes an issue (or if we're certain that it will be when our API is completed and released), then we have different ways we can improve performance. This section is mainly about using SQL JOIN statements, which we're currently not using. We discuss more performance topics in the [Performance section](#) later in the chapter.

Let's consider this GraphQL query:

```

{
  reviews {
    id
    text
    author {
      firstName
    }
  }
}

```

If we were writing an efficient SQL statement to fetch that data, we'd write:

```

SELECT reviews.id, reviews.text, users.first_name
FROM reviews
LEFT JOIN users
ON reviews.author_id = users.id

```

Let's compare this statement to what happens with our current server. We can have Knex print out statements it sends by adding a `DEBUG=knex:query` env var. When we do that and make the above GraphQL query, we see these three SQL statements:

```

$ DEBUG=knex:query npm run dev
GraphQL server running at http://localhost:4000/
SQL (1.437 ms) select * from `reviews`
SQL (0.364 ms) select * from `users` where `id` = 1
SQL (0.377 ms) select * from `users` where `id` = 1

```

There are a few issues with this:

- There are 3 queries instead of 1. (And more generally, there are `N+1` queries, where `N` is the number of reviews.)

- They all select `*` instead of just the fields needed.
- The second two are redundant (they occur because `SQLDataSource` doesn't do batching).

This probably will result in a higher load on the SQL server than the single efficient statement we wrote. It also has a higher latency, since not all of the three statements are sent at the same time—first the reviews are fetched, then the `author_id`s are used to create the rest of the statements. That's two round trips over the network from the API server to the database instead of the one trip our efficient statement took.

Let's change our code to use a JOIN like the efficient statement did. Currently, the `reviews` root Query field calls the `getReviews()` data source method:

`src/data-sources/SQL.js`

```
class SQL extends SQLDataSource {
  getReviews() {
    return this.knex
      .select('*')
      .from('reviews')
      .cache(REVIEW_TTL)
  }
  ...
}
```

We can add a `.leftJoin()` :

```
import { pick } from 'lodash'

class SQL extends SQLDataSource {
  async getReviews() {
    const reviews = await this.knex
      .select(
        'users.*',
        'users.created_at as users__created_at',
        'users.updated_at as users__updated_at',
        'reviews.*'
      )
      .from('reviews')
      .leftJoin('users', 'users.id', 'reviews.author_id')
      .cache(REVIEW_TTL)

    return reviews.map(review => ({
      ...review,
      author: {
        id: review.author_id,
        created_at: review.users__created_at,
        updated_at: review.users__updated_at,
        ...pick(review, 'first_name', 'last_name', 'email', 'photo')
      }
    })))
  }
  ...
}
```

We needed to change our `.select('*')` because both users and reviews have `created_at` and `updated_at` columns. We also needed to use `.map()` to extract out the user fields into an `author` object.

Finally, we need to stop the `Review.author` resolver from querying the database. We can do so by checking if the `author` object is already present on the review object:

`src/resolvers/Review.js`

```
export default {
  Query: ...
  Review: {
```

```

    author: (review, _, { dataSources: { db } }) =>
      review.author || db.getUser({ id: review.author_id }),
    ...
  }
}

```

Now when we run the same GraphQL query in Playground, we see this SQL statement is executed:

```

SQL (1.873 ms) select `reviews`.*, `users`.`created_at` as `users__created_at`, `users`.`updated_at` as `users__updated_at` from `reviews` left join `users` on `users`.`id` = `reviews`.`author_id`

```

Success! We went from three statements down to one. However, there are still inefficiencies. The SQL statement is overfetching in two ways:

- It's selecting all fields, whereas the GraphQL query only needed `id`, `text`, and `author.firstName`.
- It always does a JOIN, even when the GraphQL query doesn't select `Review.author`.

We can write code to address both these things—by looking through the fourth argument to resolvers, `info`, which contains information about the current GraphQL query, and seeing which fields are selected. However, it would be easier to use the [Join Monster](#) library, which does this for us.

To set it up, we create a new file to add the following information to our schema:

`src/joinMonsterAdapter.js`

```

import joinMonsterAdapt from 'join-monster-graphql-tools-adapter'

export default schema =>
  joinMonsterAdapt(schema, {
    Query: {
      fields: {
        user: {
          where: (users, args) => `${users}.id = ${args.id}`
        }
      }
    },
    Review: {
      sqlTable: 'reviews',
      uniqueKey: 'id',
      fields: {
        author: {
          sqlJoin: (reviews, users) =>
            `${reviews}.author_id = ${users}.id`
        },
        text: { sqlColumn: 'text' },
        stars: { sqlColumn: 'stars' },
        fullReview: { sqlDeps: ['text', 'stars', 'author_id'] },
        createdAt: { sqlColumn: 'created_at' },
        updatedAt: { sqlColumn: 'updated_at' }
      }
    },
    User: {
      sqlTable: 'users',
      uniqueKey: 'id',
      fields: {
        firstName: { sqlColumn: 'first_name' },
        lastName: { sqlColumn: 'last_name' },
        createdAt: { sqlColumn: 'created_at' },
        updatedAt: { sqlColumn: 'updated_at' },
        photo: { sqlDeps: ['auth_id'] }
      }
    }
  })

```

We're using the [join-monster-graphql-tools-adapter](#) package, which we need when defining our schema in SDL format via [graphql-tools](#) or Apollo Server. (We wouldn't need an adapter if we defined our schema in code with [graphql-js](#).)

We tell Join Monster:

- Which table each type corresponds to.
- Which column each field corresponds to.
- Query information for fields that involve SQL statements. For example, `Query.user` 's WHERE clause matches the `id` argument with the `id` field in the users table, and `Review.author` can be fetched with a JOIN on the users table.
- When we need it to fetch fields that aren't in the GraphQL query. For example, if `User.firstName` is in the query, it knows to fetch and return `first_name` :

```
firstName: { sqlColumn: 'first_name' },
```

But for `User.photo`, there's no photo column in the users table. So our `User.photo` resolver will run, but it needs access to the user's `auth_id` field. We need to tell Join Monster when `User.photo` is in the query, it needs to fetch `auth_id` from the database:

```
photo: { sqlDeps: ['auth_id'] }
```

We call our configuration function with a schema created by `makeExecutableSchema`, and then we pass the schema to `ApolloServer()` (whereas before we were passing `typeDefs` and `resolvers`):

`src/index.js`

```
import { makeExecutableSchema } from 'graphql-tools'

import joinMonsterAdapter from './joinMonsterAdapter'

export const schema = makeExecutableSchema({
  typeDefs,
  resolvers
})

joinMonsterAdapter(schema)

const server = new ApolloServer({
  schema,
  dataSources,
  context,
  formatError
})

...

```

We're also going to need a Knex instance, which we'll add here:

`src/data-sources/index.js`

```
import Knex from 'knex'

const knexConfig = {
  client: 'sqlite3',
  connection: {
    filename: './sql/dev.sqlite3'
  },
  useNullAsDefault: true
}

export const knex = Knex(knexConfig)
```

And lastly, we update our `Query.user` and `Query.review` resolvers:

`src/resolvers/User.js`

```
import joinMonster from 'join-monster'

import { knex } from '../data-sources/'

export default {
  Query: {
    me: ...
    user: (_, __, context, info) =>
      joinMonster(info, context, sql => knex.raw(sql), {
        dialect: 'sqlite3'
      }),
    ...
  },
  ...
}
```

src/resolvers/Review.js

```
import joinMonster from 'join-monster'

import { knex } from '../data-sources/'

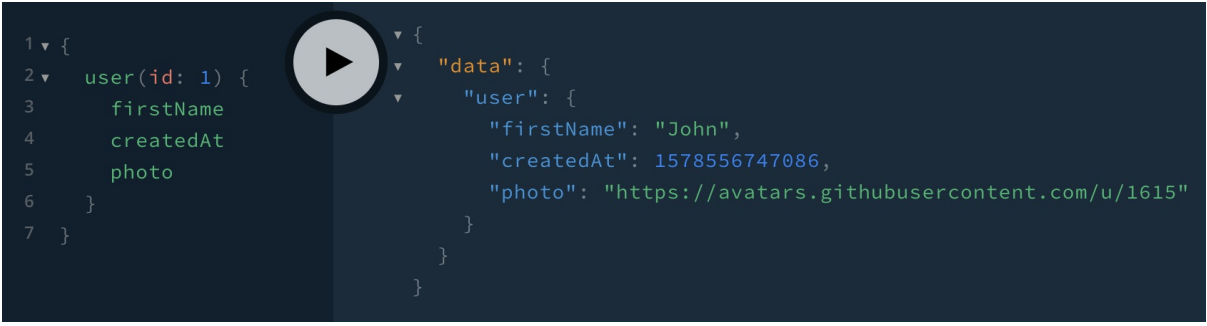
export default {
  Query: {
    reviews: (_, __, context, info) =>
      joinMonster(info, context, sql => knex.raw(sql), {
        dialect: 'sqlite3'
      })
  },
  ...
}
```

That was certainly simpler than the long `getReviews()` method we wrote! Instead, we give `joinMonster()` the `info` and `context`, and it gives us a SQL statement to run.

We also get to remove some resolvers that will be taken care of by Join Monster:

```
User.firstName
User.lastName
User.createdAt
User.updatedAt
Review.author
Review.createdAt
Review.updatedAt
```

Now when we query for a user and select `firstName`, `createdAt`, and `photo`:



```
1 {
2   user(id: 1) {
3     firstName
4     createdAt
5     photo
6   }
7 }
```

```
{
  "data": {
    "user": {
      "firstName": "John",
      "createdAt": 1578556747086,
      "photo": "https://avatars.githubusercontent.com/u/1615"
    }
  }
}
```

this SELECT statement gets run:

```
GraphQL server running at http://localhost:4000/
knex:query SELECT
knex:query  "user"."id" AS "id",
knex:query  "user"."first_name" AS "firstName",
knex:query  "user"."created_at" AS "createdAt",
```

```
knex:query  "user"."auth_id" AS "auth_id"
knex:query FROM users "user"
knex:query WHERE "user".id = 1 +16s
```

Join Monster knows to get `1` from the query argument to use in the WHERE clause, it knows to look in the users table, and it knows exactly which fields to fetch, even `auth_id`.

Here's another example of `sqlDeps` working. From the config:

```
fullReview: { sqlDeps: ['text', 'stars', 'author_id'] },
```

When we send this query:

```
{
  reviews {
    fullReview
  }
}
```

all three deps are selected:

```
knex:query SELECT
knex:query  "reviews"."id" AS "id",
knex:query  "reviews"."text" AS "text",
knex:query  "reviews"."stars" AS "stars",
knex:query  "reviews"."author_id" AS "author_id"
knex:query FROM reviews "reviews" +0ms
SQL (0.980 ms) select * from `users` where `id` = 1
SQL (0.367 ms) select * from `users` where `id` = 1
```

Join Monster [doesn't yet support](#) a joined object type as a field dependency, which is why we list `author_id` instead of `author` in `sqlDeps`, and why the `Review.fullReview` resolver still has to call `db.getUser()`.

Lastly, let's see how it handles a reviews query with `author` selected:

```
{
  reviews {
    author {
      lastName
    }
  }
}
```

```
knex:query SELECT
knex:query  "reviews"."id" AS "id",
knex:query  "author"."id" AS "author__id",
knex:query  "author"."last_name" AS "author__lastName"
knex:query FROM reviews "reviews"
knex:query LEFT JOIN users "author" ON "reviews".author_id = "author".id +3m
```

🌟 Perfect! It only fetched the fields needed and used a single statement.

## REST

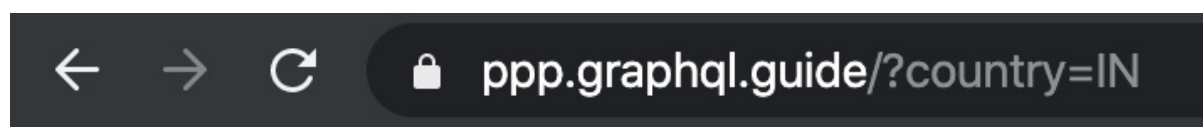
If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...rest](#))

Instead of fetching our data directly from the database, we may want to make use of our company's legacy REST services (yes, any service that doesn't speak GraphQL and support [Apollo Federation](#) is now a *legacy* service 😊😊). Or we may want to use data from third-party REST APIs. In either case, we use `RESTDataSource` to create a data source that makes REST requests.

Users of the Guide site need to be able to purchase the book, so we need to display the price to them. And let's say we wanted to make the book more affordable in locations outside of the United States where it was originally priced. [Purchasing power parity](#) (PPP) produces a conversion factor based on the actual purchasing power in a different location. For example, if the book is \$100 in the U.S., and the conversion factor for India is 0.26, then charging  $100 * 0.26 = \$26$  for the book to customers in India would make it equivalently affordable for them.

Let's add a root query field `costInCents` that returns the PPP-adjusted cost of the book. To do that, we'll need to query a PPP API. `ppp.graphql.guide` is a REST API that returns PPP information when given a country code (for example, `?country=IN` for India). We can try it out in the browser:

[ppp.graphql.guide/?country=IN](http://ppp.graphql.guide/?country=IN)



```
{
  countryCode: "IN",
  - currency: {
    exchangeRate: 74.865,
    code: "INR",
    name: "Indian rupee",
    symbol: "₹"
  },
  ppp: 18.553,
  pppConversionFactor: 0.24781940826821616
}
```

The response JSON includes `pppConversionFactor`, which combines the `ppp` value and exchange rate into a number we multiply the USD price by.

The other thing we need to figure out is how to get the country code of the client. We could look at the IP address (which is either `req.headers['x-forwarded-for']` || `req.socket.remoteAddress`) and use a GeoIP lookup API (where we send the IP address and get back a location), but the easier way is to use the Cloudflare CDN, which adds a fairly accurate `cf-ipcountry` HTTP header to all incoming requests. We can emulate this by setting the `cf-ipcountry` header in Playground.

We can check the header in our context function, and add the country code to our context object:

`src/context.js`

```
export default async ({ req }) => {
  const context = {}
```

```

...

const countryCode = req && req.headers['cf-ipcountry']
const invalidCode = ['XX', 'T1'].includes(countryCode)
if (countryCode && !invalidCode) {
  context.countryCode = countryCode
}

return context
}

```

We'll then be able to access the code from our data source, which we create by extending `RESTDataSource` from `apollo-datasource-rest`. There are five main things to know about `RESTDataSource`:

- Set `this.baseUrl` to the REST API's URL in the constructor.
- Use HTTP verb methods like `this.get(path, queryParams, options)`, `this.post()`, etc.
- It `deduplicates` REST requests.
- It caches responses from the REST API based on the responses' cache headers.
- Define a `willSendRequest()` method if you want to modify all outgoing requests—for instance, by adding an auth header:

```

class SomePrivateAPI extends RESTDataSource {
  ...

  willSendRequest(request) {
    request.headers.set('Authorization', this.context.token);
  }
}

```

Here's our implementation, using `this.baseUrl`, `this.get()`, and `this.context`:

`src/data-sources/PPP.js`

```

import { RESTDataSource } from 'apollo-datasource-rest'

export default class PPP extends RESTDataSource {
  constructor() {
    super()
    this.baseUrl = `https://ppp.graphql.guide`
  }

  async getConversionFactor() {
    const { countryCode } = this.context
    if (!countryCode) {
      return 1
    }

    const data = await this.get('/', { country: countryCode })
    return data.pppConversionFactor || 1
  }
}

```

We don't need to define `willSendRequest()` because it's a public API. We only need a single method `getConversionFactor()`, which makes a GET request of the form `/?country=[countryCode]`. It defaults to a factor of 1, which results in the full price.

Next we need to add this to our `dataSources` so we can access it from our resolvers:

`src/data-sources/index.js`

```

import PPP from './PPP'

export default () => ({
  reviews: new Reviews(db.collection('reviews')),
  users: new Users(db.collection('users')),
})

```



```

    ppp: new PPP()
  })

  export { Reviews, Users, Github, PPP }

```

And now adding our resolver:

src/resolvers/PPP.js

```

const BOOK_PRICE = 3900

export default {
  Query: {
    costInCents: async (_, __, { dataSources }) =>
      Math.round((await dataSources.ppp.getConversionFactor()) * BOOK_PRICE)
  }
}

```

src/resolvers/index.js

```

import PPP from './PPP'

const resolversByType = [Review, User, Date, Github, PPP]

...

```

Lastly, we add the `costInCents` root Query field:

src/schema/PPP.graphql

```

extend type Query {
  costInCents: Int!
}

```

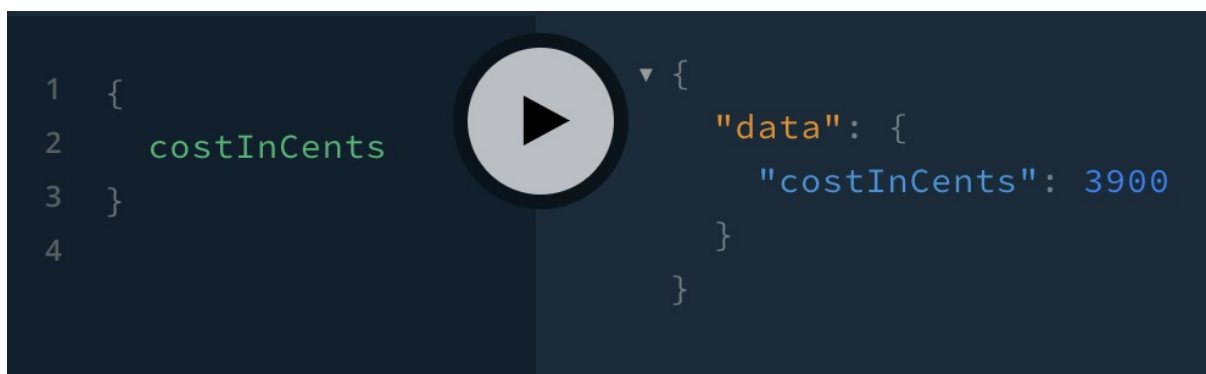
src/schema/schema.graphql

```

...
#import 'PPP.graphql'

```

Now we should be able to get 3900 in response to a `{ costInCents }` query:

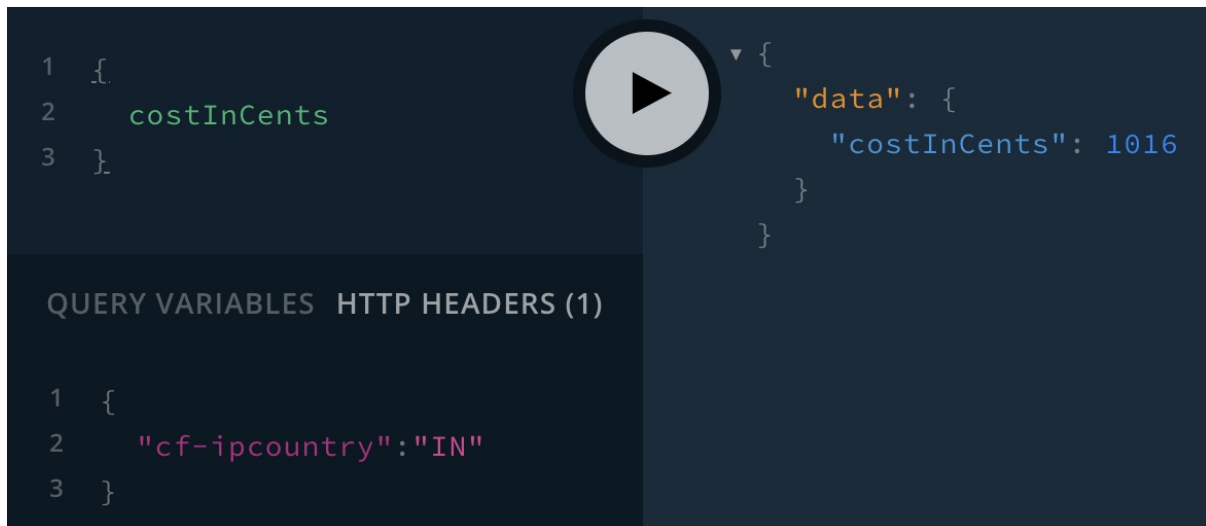


This is defaulting to the US price, since there's no header. When we add a country header, we'll see a different result:

```

{
  "cf-ipcountry": "IN"
}

```



It works! 🐛 The only thing left to check is caching. `RESTDataSource` only caches responses that contain a `Cache-Control` header. To see whether `ppp.graphql.guide` uses cache headers, we can use a command-line tool called [httpie](#) (a modern alternative to `wget`):

```
brew install httpie
```

```
$ http https://ppp.graphql.guide/?country=IN
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: application/json; charset=utf-8
cache-control: max-age=604800, public
content-length: 278
date: Wed, 05 Feb 2020 07:27:47 GMT
etag: W/"116-6RgJXuLuRrGbBbX6QFViYUXAREs"
server: now
strict-transport-security: max-age=63072000
x-now-cache: MISS
x-now-id: iad1:sf01:bxvvv-1580887666054-bbdc016271ef
x-now-trace: iad1

{
  "countryCode": "IN",
  "currency": {
    "code": "INR",
    "exchangeRate": 71.295489,
    "name": "Indian rupee",
    "symbol": "₹"
  },
  "ppp": 18.553,
  "pppConversionFactor": 0.2602268426828519
}
```

At the top of a list of headers, which includes a `cache-control` header (HTTP headers aren't case-sensitive) instructing the recipient to cache the response for 604800 seconds (one week). So now our data source *should* be saving responses to the cache, but how can we check? If we were still using [Redis as a cache](#), we could check Redis, but instead the data source is using the default in-memory cache. Without Redis, we can run [tcpdump](#) to see when our development machine makes requests to `ppp.graphql.guide`. When a country is already cached, we shouldn't see the request. In one terminal, we run this command:

```
$ sudo tcpdump "tcp[tcpflags] & (tcp-syn) != 0 and dst ppp.graphql.guide"
tcpdump: data link type PKTAP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on pktap, link-type PKTAP (Apple DLT_PKTAP), capture size 262144 bytes
```

And then we change the country header in Playground to one we haven't used, e.g., `CN` for China. On the first query, we should see this line printed:

```
04:30:18.705846 IP macbook.fios-router.home.52591 > ec2-3-210-90-207.compute-1.amazonaws.com.https: Flags [S],
seq 995289110, win 65535, options [mss 1460,nop,wscale 6,nop,nop,TS val 1101427783 ecr 0,sackOK,eol], length 0
```

which signifies a new request to `ppp.graphql.guide`. If we continue to re-issue the Playground query with the same country header, no more lines should be printed, which means the data source used the in-memory cache instead of making a request.

## GraphQL

If there's a GraphQL API that we want to use data from, we have a few options:

- If we want to include parts of the API's schema in our schema:
  - If it supports [federation](#), we should use that. For example, FaunaDB is [working on support](#), and some third-party services we use might have a GraphQL API that supports federation. And if we have control over the API (e.g., if it's one of our services), we can add support for federation.
  - We can use [schema stitching](#) if the API doesn't support federation. But unless we want a significant part of the API's schema, it may be easier to use one of the below methods instead.
- If we just want to use data from the API in our resolvers:
  - Use `GraphQLDataSource` from [apollo-datasource-graphql](#) to create a data source class. Similarly to `RESTDataSource`, we can define a `willSendRequest` method that adds an authorization header to all requests. But in our data fetching methods, instead of `this.get('path')`, we use `this.query(QUERY_DOCUMENT)`.
  - Use `graphql-request` in our resolvers to fetch data from the data source (similar to our [githubStars](#) subscription where we fetch data from GitHub's GraphQL API). While `graphql-request` is nice for extremely simple uses like `githubStars`, usually `GraphQLDataSource` is a better choice, as it's a data source class.

## Custom data source

When we've been talking about data sources, sometimes we're referring to the classes we create ( `PPP` in the below snippet), and sometimes we're referring to the parent classes that we get from an npm library and extend ( `RESTDataSource` ).

```
import { RESTDataSource } from 'apollo-datasource-rest'

class PPP extends RESTDataSource {
  ...
}
```

If there's a type of database or API for which we can't find an existing library and parent class, we can write our own! A data source parent class has most or all of the following pieces:

- Extends the `DataSource` class from the `apollo-datasource` library
- Some way of receiving information about the database or API (either a constructor parameter or an instance variable like `this.baseUrl` in `RESTDataSource` )
- An `initialize()` method that receives the context and an optional cache
- Calls lifecycle methods that can be defined by the child class, like `willSendRequest()` and `didEncounterError()` in `RESTDataSource`
- Methods for fetching data, which use `DataLoader` and/or the cache
- Methods for changing data, which might invalidate cached data

Let's see all these in a parent class called `FooDataSource` for an imaginary Foo document database. It's passed a Foo database client `dbClient`, which has these fields:

- `dbClient.connectionURI` : the URI of the database server
- `dbClient.getByIds(ids)` : given an array of IDs, returns the associated documents from the database
- `dbClient.update(id, newDoc)` : updates the document with the given `id` to the `newDoc`

```
import { DataSource } from 'apollo-datasource'
import { InMemoryLRUCache } from 'apollo-server-caching'
import DataLoader from 'dataloader'

class FooDataSource extends DataSource {
  constructor(dbClient) {
    super()
    this.db = dbClient
    this.loader = new DataLoader(ids => dbClient.getByIds(ids))
  }

  initialize({ context, cache } = {}) {
    this.context = context
    this.cache = cache || new InMemoryLRUCache()
  }

  didEncounterError(error) {
    throw error
  }

  cacheKey(id) {
    return `foo-${this.db.connectionURI}-${id}`
  }

  async get(id, { ttlInSeconds } = {}) {
    const cacheDoc = await this.cache.get(this.cacheKey(id))
    if (cacheDoc) {
      return JSON.parse(cacheDoc)
    }

    const doc = await this.loader.load(id)

    if (ttlInSeconds) {
      this.cache.set(this.cacheKey(id), JSON.stringify(doc), { ttl: ttlInSeconds })
    }

    return doc
  }

  async update(id, newDoc) {
    try {
      await this.db.update(id, newDoc)
      this.cache.delete(this.cacheKey(id))
    } catch (error) {
      this.didEncounterError(error)
    }
  }
}
```

Let's look at each part:

```
constructor(dbClient) {
  super()
  this.db = dbClient
  this.loader = new DataLoader(ids => dbClient.getByIds(ids))
}
```

The constructor saves the db client as an instance variable to be used later. It also creates an instance of `DataLoader` to use for this request (a new data source object will be created for each GraphQL request). `DataLoader` needs to know how to fetch a list of documents by their IDs. Here we're assuming the array of documents that `getByIds()`

returns is in the same order and has the same length as `ids` (a requirement of `DataLoader`); otherwise, we'd need to reorder them.

`DataLoader` is a library that does batching and memoization caching for the queries our data source makes within a single GraphQL request. **Batching** converts multiple database requests for individual documents into a single request for all the documents, and **memoization caching** deduplicates multiple requests for the same document.

```
initialize({ context, cache } = {}) {
  this.context = context
  this.cache = cache || new InMemoryLRUCache()
}
```

`initialize()` is called automatically by Apollo Server. If Apollo Server has been configured with a global cache, we use that; otherwise, we create an in-memory cache.

```
didEncounterError(error) {
  throw error
}
```

When an error occurs, we call `this.didEncounterError()`, which a child class can override.

```
cacheKey(id) {
  return `foo-${this.db.connectionURI}-${id}`
}
```

We use the `connectionURI` in the cache key to avoid collisions. A collision could occur if there were a global cache and multiple Foo data sources connected to different Foo databases, and one database had a document with the same ID as a document in another database.

```
async get(id, { ttlInSeconds } = {}) {
  const cacheDoc = await cache.get(this.cacheKey(id))
  if (cacheDoc) {
    return JSON.parse(cacheDoc)
  }

  const doc = await this.loader.load(id)

  if (ttlInSeconds) {
    cache.set(this.cacheKey(id), JSON.stringify(doc), { ttl: ttlInSeconds })
  }

  return doc
}
```

We provide a `get(id)` method to be used in resolvers, with an optional `ttlInSeconds` if the caller wants the result to be cached. First, we check if the doc is already in the cache. If it is, we parse it (cache values are always strings) and return it. Then we ask `DataLoader` to get the document. It will:

- Take all the calls to `.load(id)`. (The resolver—or other resolvers—might be calling `.get()` around the same time as this is running.)
- Deduplicate them (when `.get()` is called multiple times with the same ID).
- Put all the distinct IDs into an array for a batch request (the call to `dbClient.getByIds()` in the constructor).

Once the batch request completes, `DataLoader` returns on this line the one document we need:

```
const doc = await this.loader.load(id)
```

Then if `ttlInSeconds` was provided, we cache the document for that length of time. And finally, we return it!

```

async update(id, newDoc) {
  try {
    await this.db.update(id, newDoc)
    this.cache.delete(this.cacheKey(id))
  } catch (error) {
    this.didEncounterError(error)
  }
}

```

We provide an `update(id, newDoc)` method to be used in resolvers. After a successful update, it deletes the old document from the cache. Another possible implementation would be to overwrite the previous cache entry with `newDoc` —in this case, we'd need a value for `ttl` and could add a third argument to `update()` with a `ttlInSeconds`.

Once we have the parent class complete, we can use it by creating one or more child classes. In the case of `Foo`, we'd create one for each database, but with some data sources we might do one for each table or collection.

Here's an example child class:

```

import FooDataSource from './FooDataSource'
import { reportError } from './utils'

export default class MyFooDB extends FooDataSource {
  async updateFields(id, fields) {
    const doc = await this.get(id)
    return this.update(id, {
      ...doc,
      ...fields
    })
  }

  didEncounterError(error) {
    reportError(error)
  }
}

```

The child class overrides `didEncounterError` to use its own error reporting service instead of throwing. It adds a new method that calls the parent's `.get()` and `.update()`. When we create the data source, we give the database client to the constructor:

```

import FooClient from 'imaginary-foo-library'

import MyFooDB from './MyFooDB'

const fooClient = new FooClient({ uri: 'https://foo.graphql.guide:9001' })

const dataSources = () => ({
  myFoods: new MyFooDB(fooClient)
})

```

And now inside our resolvers, we can use `context.dataSources.myFoods` and all the methods defined in the parent class (`FooDataSource`) and child class (`MyFooDB`):

```

const resolvers = {
  Query: {
    getFoo: (_, { id }, context) => {
      context.dataSources.myFoods.get(id, { ttlInSeconds: 60 })
    },
  },
  Mutation: {
    updateFoo: async (_, { id, fields }, context) => {
      if (context.isAdmin) {
        context.dataSources.myFoods.updateFields(id, fields)
      }
    },
  },
}

```

These example resolvers use `.get()` from `FooDataSource` and `.updateFields()` from `MyFooDB`.

## Prisma

This section will be written after the release of [Prisma 2](#).

## Extended topics

- [Mocking](#)
- [Pagination](#)
- [File uploads](#)
- [Schema design](#)
- [Apollo federation](#)
- [Schema change validation](#)
- [Subscription design](#)
- [Auth options](#)
- [Security](#)
- [Caching](#)
- [Custom schema directives](#)
- [Performance](#)
- [Future](#)

This section includes miscellaneous server topics that we didn't get to in the main-line [Building](#) tutorial, the [Testing](#) sequence, the [Production](#) section, or the [data sources](#) section. Some topics are short, and some are long (yes, we know—the length of this chapter is ridiculous 😅). Most of the code will be branched off of 25, the end of the Testing sequence.

## Mocking

If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...mocking](#))

Mocking API responses—providing the client with fake (mock) data—is easy in GraphQL because we have a schema that tells us the structure of the data and the type of each field. And it's super easy with Apollo Server—we just add

```
mock: true :
```

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  mock: true
})
```

Apollo needs to know how to mock custom types, so we need a mock `Date` for our app:

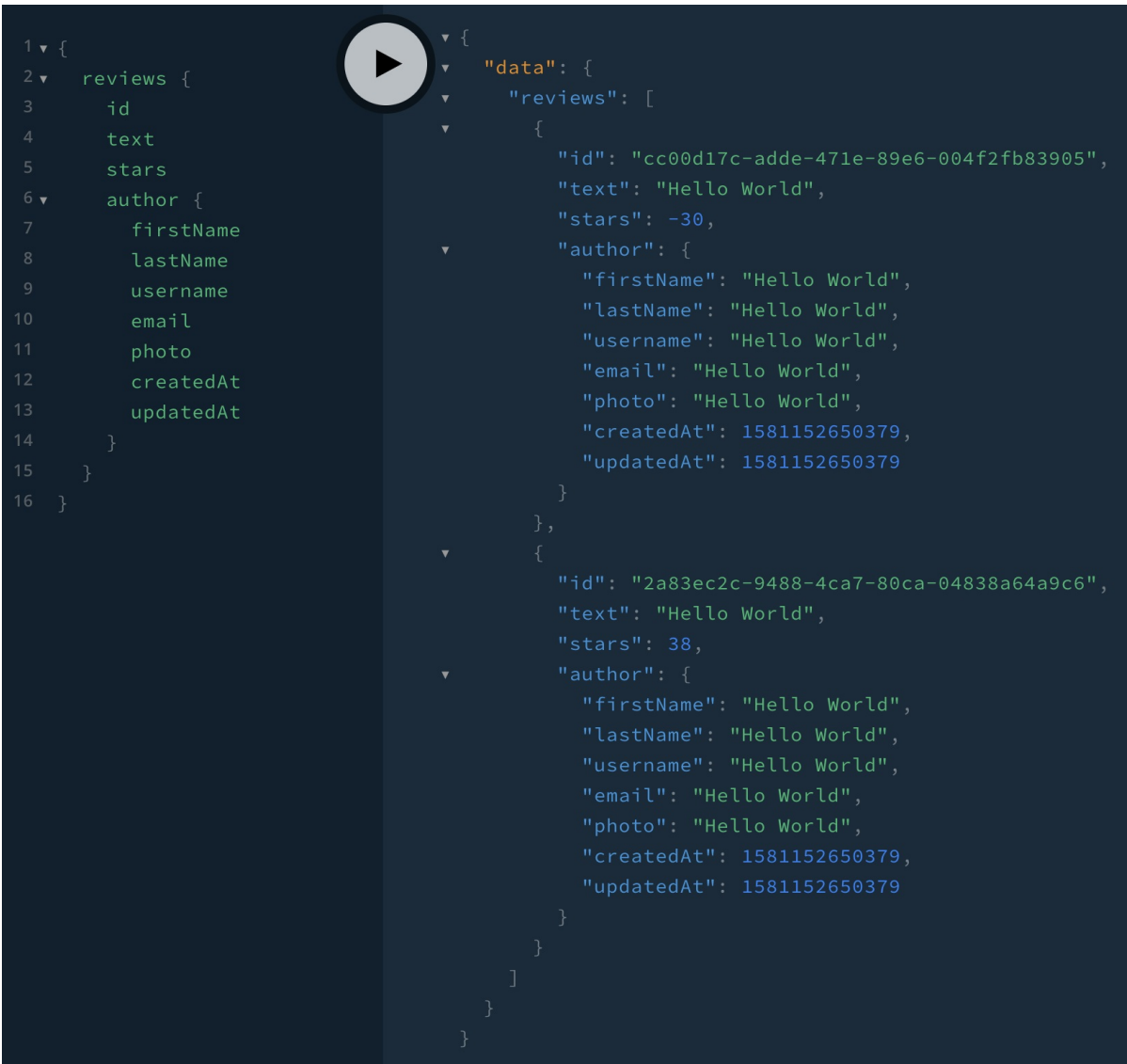
```
src/index.js
```

```
const mocks = {
  Date: () => new Date()
}

const server = new ApolloServer({
  typeDefs,
  resolvers,
  dataSources,
```

```
context,
formatError,
mocks
})
```

Now when we make a `reviews` query, all the fields we select get returned with mock data:



The screenshot shows a GraphQL query in the left pane and its JSON response in the right pane. The query is:

```
1 {
2   reviews {
3     id
4     text
5     stars
6     author {
7       firstName
8       lastName
9       username
10      email
11      photo
12      createdAt
13      updatedAt
14    }
15  }
16 }
```

The response is:

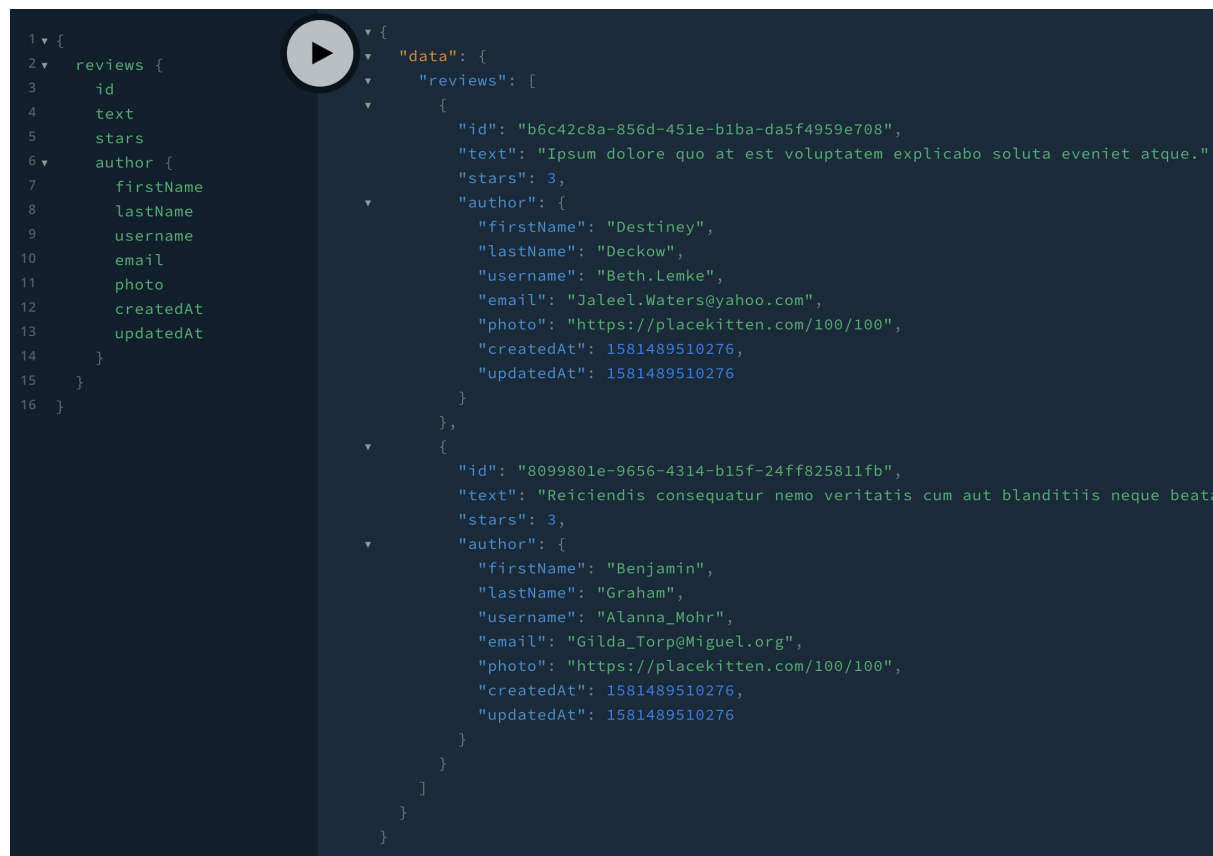
```
{
  "data": {
    "reviews": [
      {
        "id": "cc00d17c-adde-471e-89e6-004f2fb83905",
        "text": "Hello World",
        "stars": -30,
        "author": {
          "firstName": "Hello World",
          "lastName": "Hello World",
          "username": "Hello World",
          "email": "Hello World",
          "photo": "Hello World",
          "createdAt": 1581152650379,
          "updatedAt": 1581152650379
        }
      },
      {
        "id": "2a83ec2c-9488-4ca7-80ca-04838a64a9c6",
        "text": "Hello World",
        "stars": 38,
        "author": {
          "firstName": "Hello World",
          "lastName": "Hello World",
          "username": "Hello World",
          "email": "Hello World",
          "photo": "Hello World",
          "createdAt": 1581152650379,
          "updatedAt": 1581152650379
        }
      }
    ]
  }
}
```

If we want them to look more like real data, we can use the `casual` library for fake data generation:

```
import casual from 'casual'

const mocks = {
  Date: () => new Date(),
  Review: () => ({
    text: casual.sentence,
    stars: () => casual.integer(0, 5)
  }),
  User: () => ({
    firstName: casual.first_name,
    lastName: casual.last_name,
    username: casual.username,
    email: casual.email,
    photo: `https://placekitten.com/100/100`
  })
}
```





To make the results array have a variable number of results (the default is two items for all lists), we could add this to make it return between 0 and 3 items:

```
import { ApolloServer, MockList } from 'apollo-server'

const mocks = {
  ...
  Query: () => ({
    reviews: () => new MockList([0, 3])
  })
}
```

If we created a new app with mocking, and then we wanted to start writing real resolvers, we could add `resolvers` and `mockEntireSchema: false`:

```
const server = new ApolloServer({
  typeDefs,
  mocks,
  resolvers,
  mockEntireSchema: false
})
```

Then our resolvers would be used first, and mocks would be used for all the fields for which we hadn't yet written resolvers.

We can also mock a schema written in a different language than JavaScript or a schema from a third-party GraphQL API. First we download `graphql-cli`, and then we use it to download the target API's schema:

```
$ npm i -g graphql-cli
$ graphql get-schema -e https://api.spacex.land/graphql -o schema.json
```

Then we start a simple Apollo Server:

```
const { buildClientSchema } = require('graphql')
const introspectionResult = require('./schema.json')
const { ApolloServer } = require('apollo-server')

const schema = buildClientSchema(introspectionResult.data)

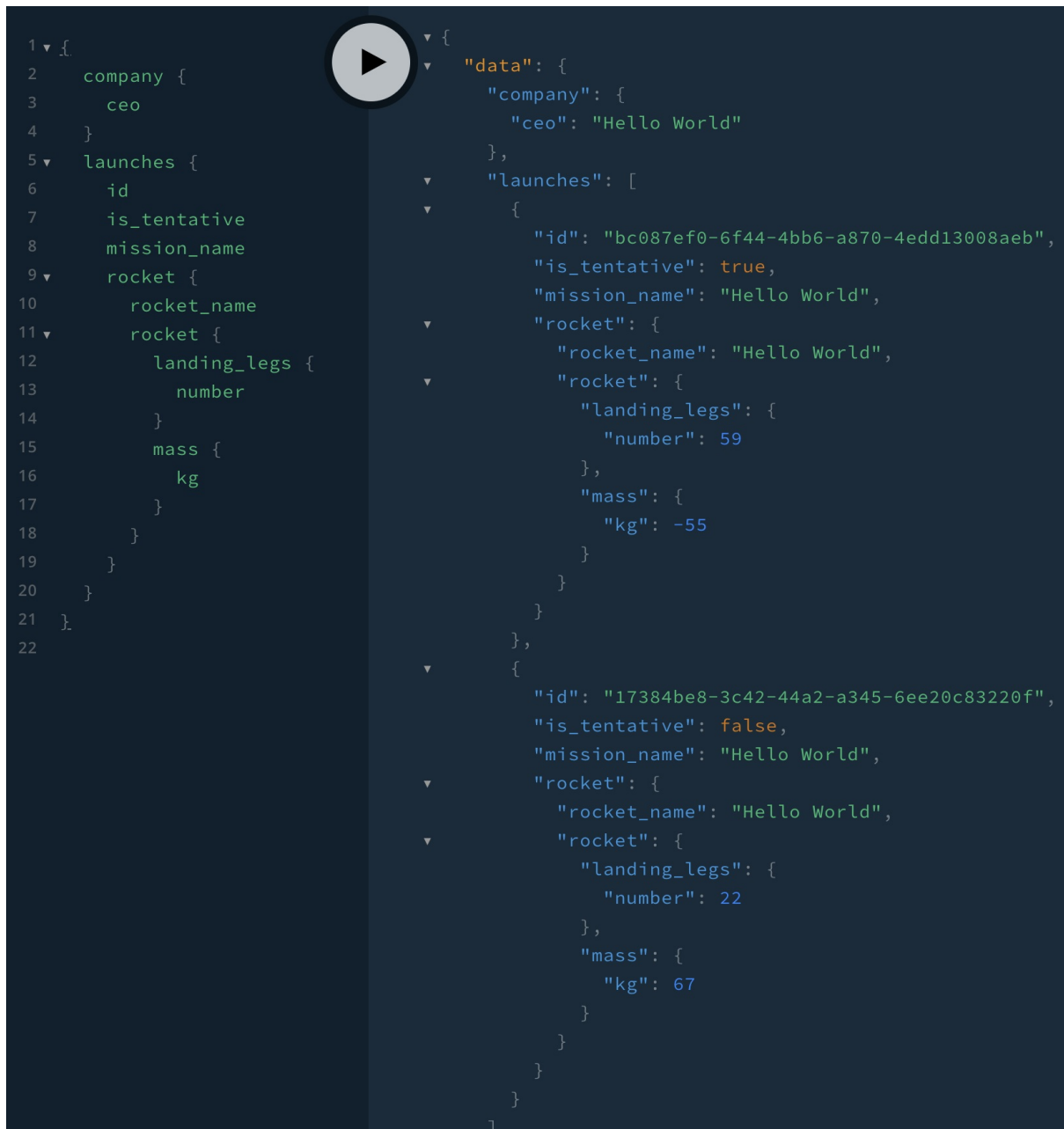
const server = new ApolloServer({
  schema,
  mocks: true
})

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

To test it, we do:

```
$ git clone https://github.com/GraphQLGuide/mock-external-schema.git
$ cd mock-external-schema
$ npm install
$ npm start
```

And we open [localhost:4000](http://localhost:4000) to issue a query:



## Pagination

- [Offset-based](#)
- [Cursors](#)
  - [after an ID](#)
  - [Relay cursor connections](#)

Pagination is the general term for requesting chunks of a list of data instead of the whole list, because requesting the whole list would take too much time or resources. In [Chapter 6: Paginating](#), we covered different types of pagination from the client's perspective. In this section, we'll cover them from the server's perspective: Defining the schema and writing code that fetches the requested chunk of data from the database.

These are the main types of pagination:

- *Offset-based*: Request a chunk at an offset from the beginning of the list.

- **Pages:** Request Nth page of a certain size. For instance, `page: 3, size: 10` would be items 21-30.
- **Skip & limit:** Request *limit* items after skipping *skip* items. For instance `skip: 40, limit: 20` would be items 41-60.
- **Cursor-based:** Request a chunk before or after a *cursor*. Conceptually, a cursor is a pointer to a location in a query's result set. There's a range of ways to implement it, both in terms of what arguments are used and how the schema looks. Here are a couple options:
  - **after an ID:** Request *limit* items *after* some sortable field, like `id`—in MongoDB, ObjectId's sort by the time they were created, like a `createdAt` timestamp. This is the simplified, cursor-like system used in [Chapter 6: Cursors](#). For instance `after: '5d3202c4a044280cac1e2f60', limit: 10` would be the 10 items after that `id`.
  - **Relay cursor connections:** Request the *first* N items *after* an opaque cursor (or *last* N items *before* a cursor). For instance, `first: 10, after: 'abcabcabc'`, where `'abcabcabc'` contains an encoded result set location.

In Chapter 6, we used `[id]:[sort order]` as the cursor format (like `'100:createdAt_DESC'`). However, it's best practice for the client to treat cursors as opaque strings, and that's usually facilitated by the server Base64-encoding the string. So the server would return `'MTAwOmNyZWZ0ZW9ERVND'` as the cursor instead of `'100:createdAt_DESC'`.

The downsides to offset-based are:

- When the result set changes (items added or removed), we might miss or get duplicate results. (We discuss this scenario in [Chapter 6: skip & limit](#).)
- The performance of a `LIMIT x OFFSET y` query does not scale well for large data sets in many databases, including PostgreSQL, MySQL, and MongoDB. (Note that depending on the flexibility of our collection structure, we might be able to use [the bucket pattern](#) in MongoDB to scale this query well.)

The downsides to cursor-based are:

- We can't jump ahead, for example, from page 1 to page 5.
- The implementation is a little more complex.

In [Offset-based](#), we'll implement skip & limit. Then in [Cursor-based](#), we'll implement [after an ID](#) and [Relay cursor connections](#).

## Offset-based

If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...pagination](#))

In skip & limit, we have three arguments: `skip`, `limit`, and `orderBy`. Let's update the schema first, then the resolver, and lastly the data sources.

For `orderBy`, we need a new enum type. The `skip` and `limit` arguments are integers. We can set default values for each so that we can make each argument nullable.

Here's the current `reviews` Query:

[src/schema/Review.graphql](#)

```
extend type Query {
  reviews: [Review!]!
}
```

Here we add the arguments:

```
enum ReviewOrderBy {
  createdAt_ASC
  createdAt_DESC
}
```

```

extend type Query {
  reviews(skip: Int, limit: Int, orderBy: ReviewOrderBy): [Review!]!
}

```

The convention for enum values is `ALL_CAPS`, but `createdAt_ASC` makes it more clear than `CREATED_AT_ASC` that it's sorting by the `Review.createdAt` field. The subsequent underscore and all-caps `ASC/DESC` still demonstrate they're enum values.

Learn the rules so you know how to break them properly. —The Dalai Lama's Fifth Rule of Living

Our resolver is currently very simple:

`src/resolvers/Review.js`

```

export default {
  Query: {
    reviews: (_, __, { dataSources }) => dataSources.reviews.all(),
    ...
  }
}

```

We need to add the arguments and check them. GraphQL execution adequately checks `orderBy` (so we know it will either be the string `'createdAt_DESC'` or `'createdAt_ASC'`), but it only checks that `skip` and `limit` are integers. We also need to make sure they're not invalid or restricted values. It doesn't make sense for `skip` to be less than 0, nor for `limit` to be less than 1. We'll also prevent large values of `limit` to protect against [denial of service attacks](#).

```

const MAX_PAGE_SIZE = 100

export default {
  Query: {
    reviews: (
      _/
      { skip = 0, limit = 10, orderBy = 'createdAt_DESC' },
      { dataSources }
    ) => {
      const errors = {}

      if (skip < 0) {
        errors.skip = `must be non-negative`
      }

      if (limit < 1) {
        errors.limit = `must be positive`
      }

      if (limit > MAX_PAGE_SIZE) {
        errors.limit = `cannot be greater than ${MAX_PAGE_SIZE}`
      }

      if (!isEmpty(errors)) {
        throw new InputError({ review: errors })
      }

      return dataSources.reviews.getPage({ skip, limit, orderBy })
    },
    ...
  }
}

```

Lastly, call a new data source method `getPage`, which we'll define next. Here's our old `.all()` method:

`src/data-sources/Reviews.js`

```

export default class Reviews extends MongoDataSource {
  all() {
    return this.collection.find().toArray()
  }
}

```

```

    }
    ...
  }

```

We replace it with:

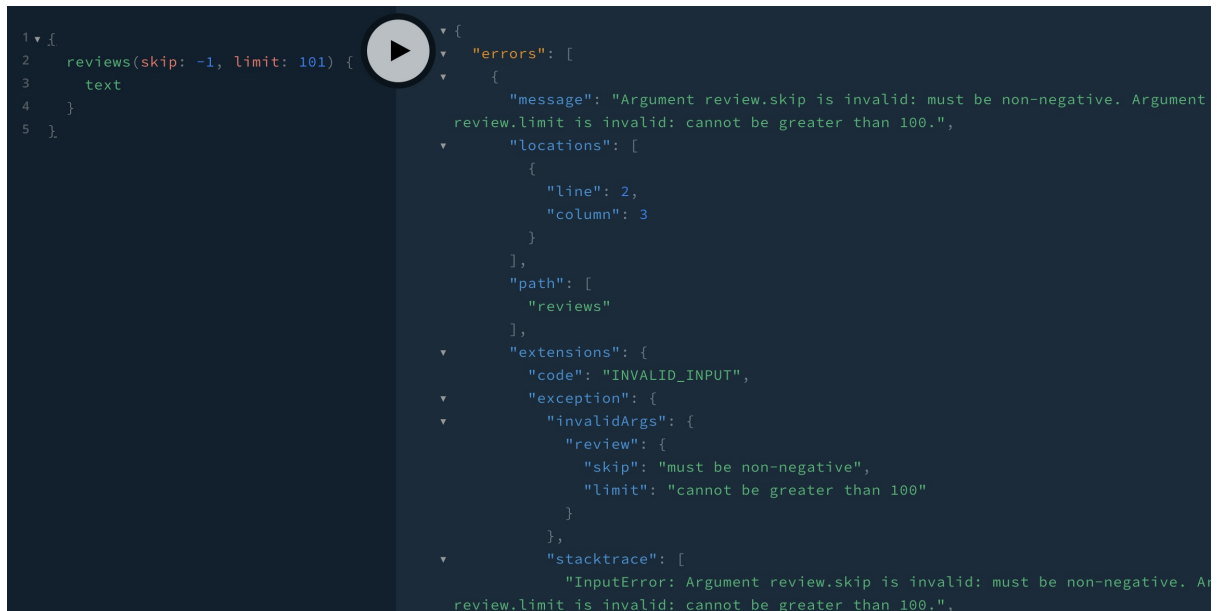
```

export default class Reviews extends MongoDataSource {
  getPage({ skip, limit, orderBy }) {
    return this.collection
      .find()
      .sort({ _id: orderBy === 'createdAt_DESC' ? -1 : 1 })
      .skip(skip)
      .limit(limit)
      .toArray()
  }
  ...
}

```

`_id` is an ObjectId, so sorting by `_id` is equivalent to sorting by a `createdAt` timestamp.

Let's first test the error case in Playground:



```

1 reviews(skip: -1, limit: 101) {
2   text
3 }
4
5

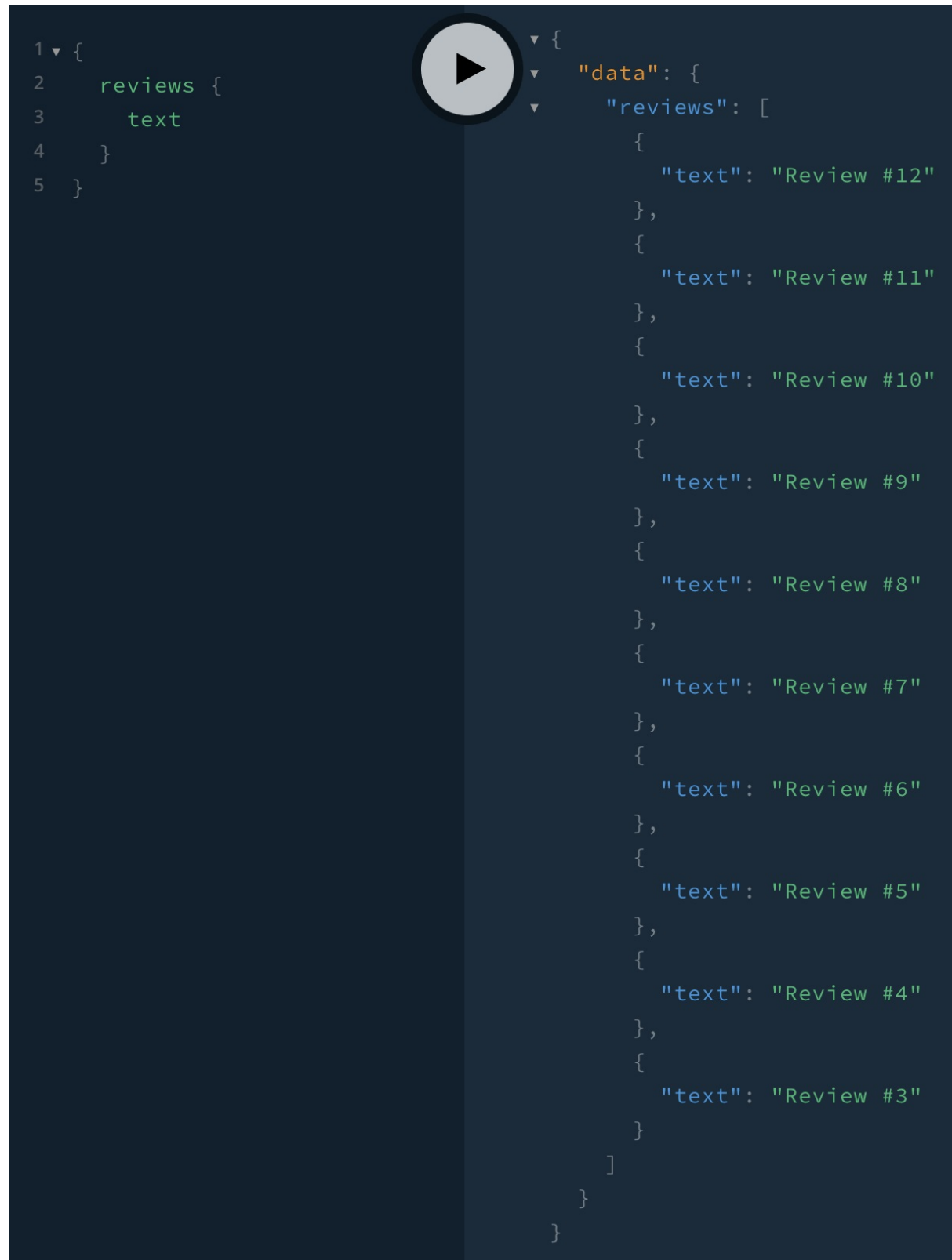
```

```

{
  "errors": [
    {
      "message": "Argument review.skip is invalid: must be non-negative. Argument review.limit is invalid: cannot be greater than 100.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "reviews"
      ],
      "extensions": {
        "code": "INVALID_INPUT",
        "exception": {
          "invalidArgs": {
            "review": {
              "skip": "must be non-negative",
              "limit": "cannot be greater than 100"
            }
          }
        },
        "stacktrace": [
          "InputError: Argument review.skip is invalid: must be non-negative. Argument review.limit is invalid: cannot be greater than 100.",
          ...
        ]
      }
    }
  ]
}

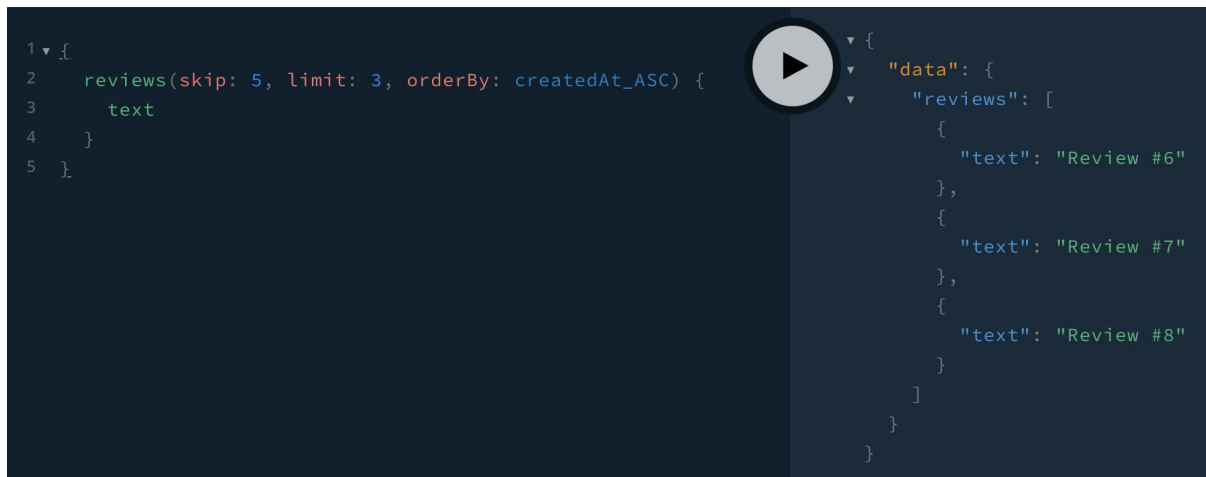
```

And with default arguments, we see the most recent 10 reviews:



```
1 {  
2   reviews {  
3     text  
4   }  
5 }  
  
{  
  "data": {  
    "reviews": [  
      {  
        "text": "Review #12"  
      },  
      {  
        "text": "Review #11"  
      },  
      {  
        "text": "Review #10"  
      },  
      {  
        "text": "Review #9"  
      },  
      {  
        "text": "Review #8"  
      },  
      {  
        "text": "Review #7"  
      },  
      {  
        "text": "Review #6"  
      },  
      {  
        "text": "Review #5"  
      },  
      {  
        "text": "Review #4"  
      },  
      {  
        "text": "Review #3"  
      }  
    ]  
  }  
}
```

And with `skip: 5, limit: 3, orderBy: createdAt_ASC`, we see the 6th through 8th reviews:



## Cursors

There are a number of ways to do cursor-based pagination:

- `after` an ID: Use three arguments to support cursor-like pagination for queries sorted by a single field ( `createdAt` ).
- `first/after` & `last/before` : `first` and `last` are equivalent to `limit` , and `after/before` is the cursor. These are added as arguments, but the client has to get the cursor from the server, which requires adding a `cursor` field to the schema. We can do this a few ways:
  1. Add `cursor` to each object.
  2. Have each paginated query return a `startCursor` , an `endCursor` , and `nodes` .
  3. Use Relay cursor connections, where the paginated query returns edges, which each contain a `cursor` and a `node` .

In this section, we will implement `after` an ID and Relay cursor connections.

#1 would have `Review.cursor` :

```
type Review {
  id: ID!
  author: User!
  text: String!
  stars: Int
  fullReview: String!
  createdAt: Date!
  updatedAt: Date!
  cursor: String
}

enum ReviewOrderBy {
  createdAt_ASC
  createdAt_DESC
}

extend type Query {
  reviews(first: Int, after: String): [Review!]!
  get(id: ID!): Review
}
```

One downside to this approach is the cursor isn't really part of a Review's data. For instance, it's not applicable when we do a `get` Query to fetch a single Review by ID.

#2 would fix that issue, since the cursor is no longer a Review field:

```
type ReviewsResult {
```



```

    nodes: [Review!]!
    startCursor: String!
    endCursor: String!
  }

  extend type Query {
    reviews(first: Int, after: String, last: Int, before: String): ReviewsResult!
    get(id: ID!): Review
  }

```

We could also add information about the data set—the total number of items and whether there are more items available to query:

```

type ReviewsResult {
  nodes: [Review!]!
  startCursor: String!
  endCursor: String!
  totalCount: Int!
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
}

```

#3 has the most involved schema, which we'll go over in [the last section](#):

```

type ReviewEdge {
  cursor: String!
  node: Review
}

type PageInfo {
  startCursor: String!
  endCursor: String!
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
}

type ReviewsConnection {
  edges: [ReviewEdge]
  pageInfo: PageInfo!
  totalCount: Int!
}

extend type Query {
  reviews(first: Int, after: String, last: Int, before: String): ReviewsConnection!
  get(id: ID!): Review
}

```

The main two benefits to #3 over #2 are:

- We have the cursor of every object—not just the start and end cursors—so we can request the next page starting at any location in the list.
- We can add more information to the edge. For instance if we had a social platform with a paginated `User.friends` field returning a `FriendsConnection` with `edges: [FriendEdge]`, a `FriendEdge` could include:

```

type FriendEdge {
  cursor: String!
  node: Friend
  becameFriendsOn: Date
  mutualFriends: [Friends]
  photosInCommon: [Photo]
}

```

**after an ID**

If you're jumping in here, `git checkout pagination_0.1.0` (tag [pagination\\_0.1.0](#), or compare [pagination...pagination2](#))

In this section we'll do a limited cursor-like pagination with these three arguments:

`src/schema/Review.graphql`

```
extend type Query {
  reviews(after: ID!, limit: Int, orderBy: ReviewOrderBy): [Review!]!
}
```

The only change from [skip & limit](#) is instead of *skipping* a number of results, we return those *after* an ID. In our resolver, we change `skip` -> `after` and remove `skip`'s error checking:

`src/resolvers/Review.js`

```
export default {
  Query: {
    reviews: (
      // { after, limit = 10, orderBy = 'createdAt_DESC' },
      { dataSources }
    ) => {
      const errors = {}

      if (limit < 0) {
        errors.limit = `must be non-negative`
      }

      if (limit > MAX_PAGE_SIZE) {
        errors.limit = `cannot be greater than ${MAX_PAGE_SIZE}`
      }

      if (!isEmpty(errors)) {
        throw new InputError({ review: errors })
      }

      return dataSources.reviews.getPage({ after, limit, orderBy })
    },
    ...
  }
}
```

We could also check whether `after` is a valid `ObjectId` (as we do in the `query.user` resolver).

In the data source, if `after` is provided (it's optional), we filter using either `$lt` or `$gt` (less than / greater than):


`src/data-sources/Review.js`

```
import { ObjectId } from 'mongodb'

export default class Reviews extends MongoDataSource {
  getPage({ after, limit, orderBy }) {
    const filter = {}
    if (after) {
      const afterId = ObjectId(after)
      filter._id =
        orderBy === 'createdAt_DESC' ? { $lt: afterId } : { $gt: afterId }
    }

    return this.collection
      .find(filter)
      .sort({ _id: orderBy === 'createdAt_DESC' ? -1 : 1 })
      .limit(limit)
      .toArray()
  }
  ...
}
```

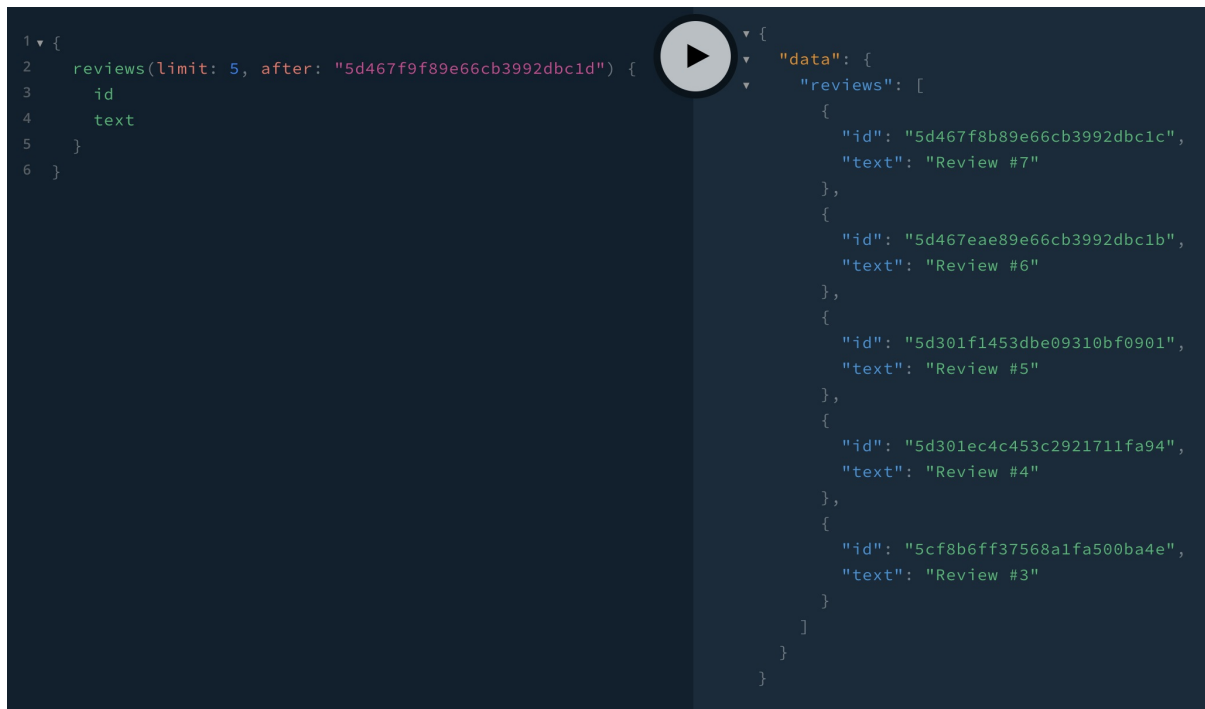
To test, first let's get the first 5 reviews with their IDs:



```
1 {
2   reviews(limit: 5) {
3     id
4     text
5   }
6 }
```

```
{
  "data": {
    "reviews": [
      {
        "id": "5d4680a889e66cb3992dbc21",
        "text": "Review #12"
      },
      {
        "id": "5d4680a189e66cb3992dbc20",
        "text": "Review #11"
      },
      {
        "id": "5d467fe889e66cb3992dbc1f",
        "text": "Review #10"
      },
      {
        "id": "5d467fb889e66cb3992dbc1e",
        "text": "Review #9"
      },
      {
        "id": "5d467f9f89e66cb3992dbc1d",
        "text": "Review #8"
      }
    ]
  }
}
```

Then we take the last ID and use it for the `after` argument:



```

1 {
2   reviews(limit: 5, after: "5d467f9f89e66cb3992dbc1d") {
3     id
4     text
5   }
6 }

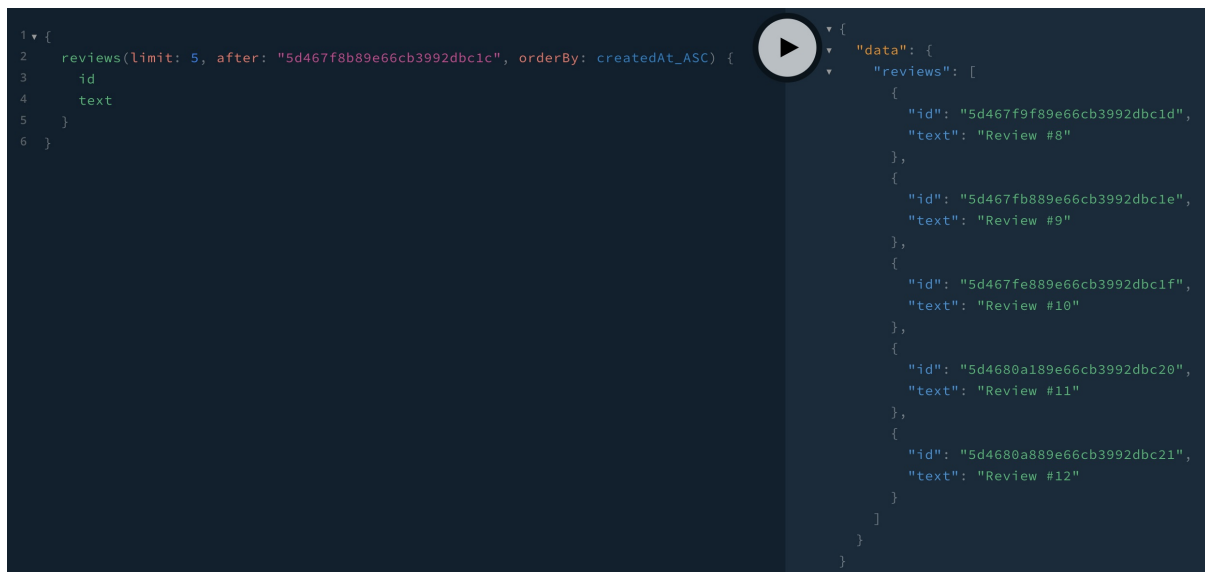
```

```

{
  "data": {
    "reviews": [
      {
        "id": "5d467f8b89e66cb3992dbc1c",
        "text": "Review #7"
      },
      {
        "id": "5d467eae89e66cb3992dbc1b",
        "text": "Review #6"
      },
      {
        "id": "5d301f1453dbe09310bf0901",
        "text": "Review #5"
      },
      {
        "id": "5d301ec4c453c2921711fa94",
        "text": "Review #4"
      },
      {
        "id": "5cf8b6ff37568a1fa500ba4e",
        "text": "Review #3"
      }
    ]
  }
}

```

It works! If we wanted to paginate the other way from review #7, we would switch the `orderBy` :



```

1 {
2   reviews(limit: 5, after: "5d467f8b89e66cb3992dbc1c", orderBy: createdAt_ASC) {
3     id
4     text
5   }
6 }

```

```

{
  "data": {
    "reviews": [
      {
        "id": "5d467f9f89e66cb3992dbc1d",
        "text": "Review #8"
      },
      {
        "id": "5d467fb889e66cb3992dbc1e",
        "text": "Review #9"
      },
      {
        "id": "5d467fe889e66cb3992dbc1f",
        "text": "Review #10"
      },
      {
        "id": "5d4680a189e66cb3992dbc20",
        "text": "Review #11"
      },
      {
        "id": "5d4680a889e66cb3992dbc21",
        "text": "Review #12"
      }
    ]
  }
}

```

## Relay cursor connections

If you're jumping in here, `git checkout pagination2_0.1.0` (tag [pagination2\\_0.1.0](#), or compare [pagination2...pagination3](#))

Relay cursor connections are defined by the [Relay Cursor Connections spec](#). It specifies a standard way of implementing cursor pagination so that different clients and tools (like the Relay client library) can depend on that specific schema structure. Its benefits over other cursor structures are listed at the end of the [Cursors](#) section above. Its cost is a more complex schema, like this one:

`src/schema/Review.graphql`

```

type ReviewEdge {
  cursor: String!
  node: Review
}

```

```

}

type PageInfo {
  startCursor: String!
  endCursor: String!
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
}

type ReviewsConnection {
  edges: [ReviewEdge]
  pageInfo: PageInfo!
  totalCount: Int!
}

extend type Query {
  reviews(first: Int, after: String, last: Int, before: String): ReviewsConnection!
}

```

Including both `first/after` and `last/before` is optional—according to the spec, only one is required. Also, we can add fields—for instance, `totalCount` isn't in the spec—and add arguments to `Query.reviews` (for instance, filtering and sorting arguments). Common added arguments include a `filterBy` object type and `orderBy`, which can be an enum as we've been doing or a list (for example `orderBy: [stars_DESC, createdAt_ASC]`). Let's do just `first/after`, `orderBy`, and a single filter field—`stars`:

```

extend type Query {
  reviews(first: Int, after: String, orderBy: ReviewOrderBy, stars: Int): ReviewsConnection!
}

```

For implementing the resolver, first we check arguments:

`src/resolvers/Review.js`

```

export default {
  Query: {
    reviews: async (
      { first = 10, after, orderBy = 'createdAt_DESC', stars },
      { dataSources }
    ) => {
      const errors = {}

      if (first !== undefined && first < 1) {
        errors.first = `must be non-negative`
      }

      if (first > MAX_PAGE_SIZE) {
        errors.first = `cannot be greater than ${MAX_PAGE_SIZE}`
      }

      if (stars !== undefined && ![0, 1, 2, 3, 4, 5].includes(stars)) {
        errors.stars = `must be an integer between 0 and 5, inclusive`
      }

      if (!isEmpty(errors)) {
        throw new InputError({ review: errors })
      }

      // ... TODO

      return {
        edges,
        pageInfo: {
          startCursor,
          endCursor,
          hasNextPage,
          hasPreviousPage
        },
        totalCount
      }
    }
  }
}

```

```

    }
  },
  ...
}

```

Then, after some work (which will include one or more calls to `dataSources.reviews.*`), we return an object matching the `ReviewsConnection` in our schema:

```

type ReviewsConnection {
  edges: [ReviewEdge]
  pageInfo: PageInfo!
  totalCount: Int!
}

```

Here's how to construct that object:

```

import { encodeCursor } from '../util/pagination'

export default {
  Query: {
    reviews: async (
      _,
      { first = 10, after, orderBy = 'createdAt_DESC', stars },
      { dataSources }
    ) => {
      ...

      const {
        reviews,
        hasNextPage,
        hasPreviousPagePromise
      } = await dataSources.reviews.getPage({ first, after, orderBy, stars })

      const edges = reviews.map(review => ({
        cursor: encodeCursor(review),
        node: review
      }))

      return {
        edges,
        pageInfo: {
          startCursor: encodeCursor(reviews[0]),
          endCursor: encodeCursor(reviews[reviews.length - 1]),
          hasNextPage,
          hasPreviousPage: hasPreviousPagePromise
        },
        totalCount: dataSources.reviews.getCount({ stars })
      }
    },
  },
}

```

`dataSources.reviews.getPage()` returns an object with three things. We use `reviews` to create the edges and cursors. Each field returned from a resolver can either be a value or a Promise that resolves to a value (Apollo Server will resolve the Promise for us if that field is selected in the query). Instead of a boolean for `hasPreviousPage`, we return a Promise. And for `totalCount`, we call a new data source method `getCount()`:

`src/data-sources/Reviews.js`

```

export default class Reviews extends MongoDataSource {
  getCount(filter) {
    return this.collection.find(filter).count()
  }

  ...
}

```

The code for `getPage()` is a bit complex. We'll make three database queries to fetch the list of reviews and determine whether there are next and previous pages:

```
import { decodeCursor } from '../util/pagination'

export default class Reviews extends MongoDataSource {
  getPage({ first, after, orderBy, stars }) {
    const isDescending = orderBy === 'createdAt_DESC'
    const filter = {}
    const prevFilter = {}

    if (after) {
      const afterId = decodeCursor(after)
      filter._id = isDescending ? { $lt: afterId } : { $gt: afterId }
      prevFilter._id = isDescending ? { $gte: afterId } : { $lte: afterId }
    }

    if (stars) {
      filter.stars = stars
    }

    const sort = { _id: isDescending ? -1 : 1 }

    const reviewsPromise = this.collection
      .find(filter)
      .sort(sort)
      .limit(first)
      .toArray()

    const hasNextPagePromise = this.collection
      .find(filter)
      .sort(sort)
      .skip(first)
      .hasNext()

    const hasPreviousPagePromise =
      !!after &&
      this.collection
        .find(prevFilter)
        .sort(sort)
        .hasNext()

    return { reviewsPromise, hasNextPagePromise, hasPreviousPagePromise }
  }
  ...
}
```

The reviews query has:

```
.limit(first)
.toArray()
```

Whereas to see if there's a next item, we do:

```
.skip(first)
.hasNext()
```

And to check if there's a previous item, we use the opposite `filter` ( `$gte` and `$lte` are greater/less than or equal to) and `hasNext()` :

```
prevFilter._id = isDescending ? { $gte: afterId } : { $lte: afterId }
...
this.collection
  .find(prevFilter)
  .sort(sort)
  .hasNext()
```

If the number of database queries became a performance problem, we could remove the need for the second by changing `.limit(first)` in the reviews query to `.limit(first + 1)`. Then, if we receive `first + 1` results, we know there's a next page:

```
...

const reviews = await this.collection
  .find(filter)
  .sort(sort)
  .limit(first + 1)
  .toArray()

const hasNextPage = reviews.length > first
if (hasNextPage) {
  reviews.pop()
}

const hasPreviousPagePromise =
  !!after &&
  this.collection
    .find(prevFilter)
    .sort(sort)
    .hasNext()

return { reviews, hasNextPage, hasPreviousPagePromise }
}
```

We do `reviews.pop()` to take the extra last review (which the client didn't request) off the list.

Now we have a new issue: Our latency has gone up, since we're making two database queries in serial ( `await` ing one before starting the other) instead of three queries in parallel (initiating them all at the same time). To fix this, we can create the `hasPreviousPagePromise` before the `await`:

```
const hasPreviousPagePromise =
  !!after &&
  this.collection
    .find(prevFilter)
    .sort(sort)
    .hasNext()

const reviews = await this.collection
  .find(filter)
  .sort(sort)
  .limit(first + 1)
  .toArray()

const hasNextPage = reviews.length > first
if (hasNextPage) {
  reviews.pop()
}

return { reviews, hasNextPage, hasPreviousPagePromise }
}
```

If, however, we were more concerned with database load than latency, and clients frequently made reviews queries without selecting `query.reviews.pageInfo.hasPreviousPage`, then we could make those queries only trigger a single database query. We can do this by moving `hasPreviousPage` from a property in an object returned by the `query.reviews` resolver (what we're currently doing) to a `PageInfo.hasPreviousPage` resolver:

```
...

const getHasPreviousPage = () =>
  !!after &&
  this.collection
```



```

      .find(prevFilter)
      .sort(sort)
      .hasNext()

    return { reviews, hasNextPage, getHasPreviousPage }
  }

```

And then we update the resolvers:

`src/resolvers/Review.js`

```

export default {
  Query: {
    reviews: async (
      { first = 10, after, orderBy = 'createdAt_DESC', stars },
      { dataSources }
    ) => {
      ...

      const {
        reviews,
        hasNextPage,
        getHasPreviousPage
      } = await dataSources.reviews.getPage({ first, after, orderBy, stars })

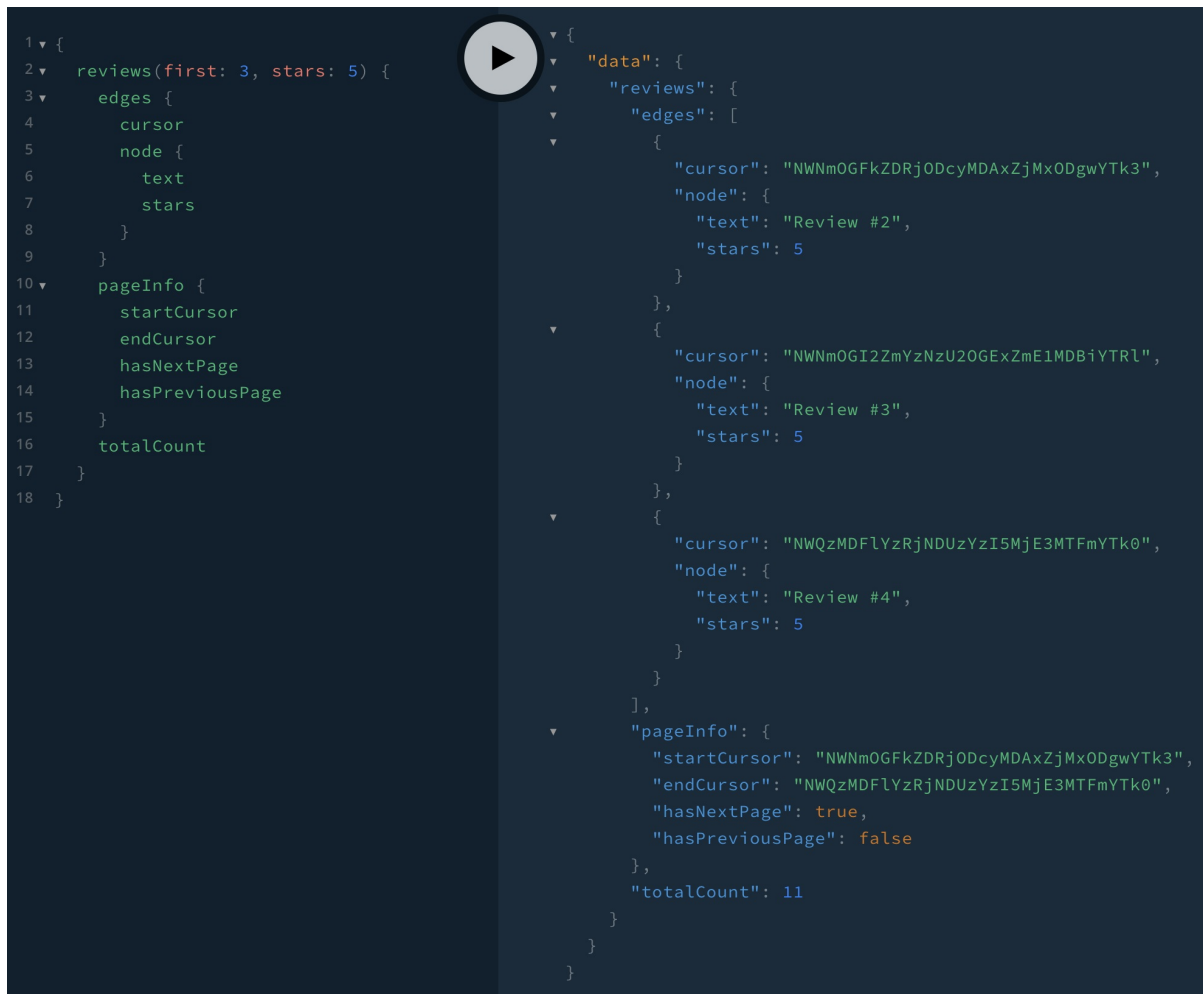
      const edges = reviews.map(review => ({
        cursor: encodeCursor(review),
        node: review
      })))

      return {
        edges,
        pageInfo: {
          startCursor: encodeCursor(reviews[0]),
          endCursor: encodeCursor(reviews[reviews.length - 1]),
          hasNextPage,
          getHasPreviousPage
        },
        totalCount: dataSources.reviews.getCount({ stars })
      }
    },
  },
  PageInfo: {
    hasPreviousPage: ({ getHasPreviousPage }) => getHasPreviousPage()
  },
  ...
}

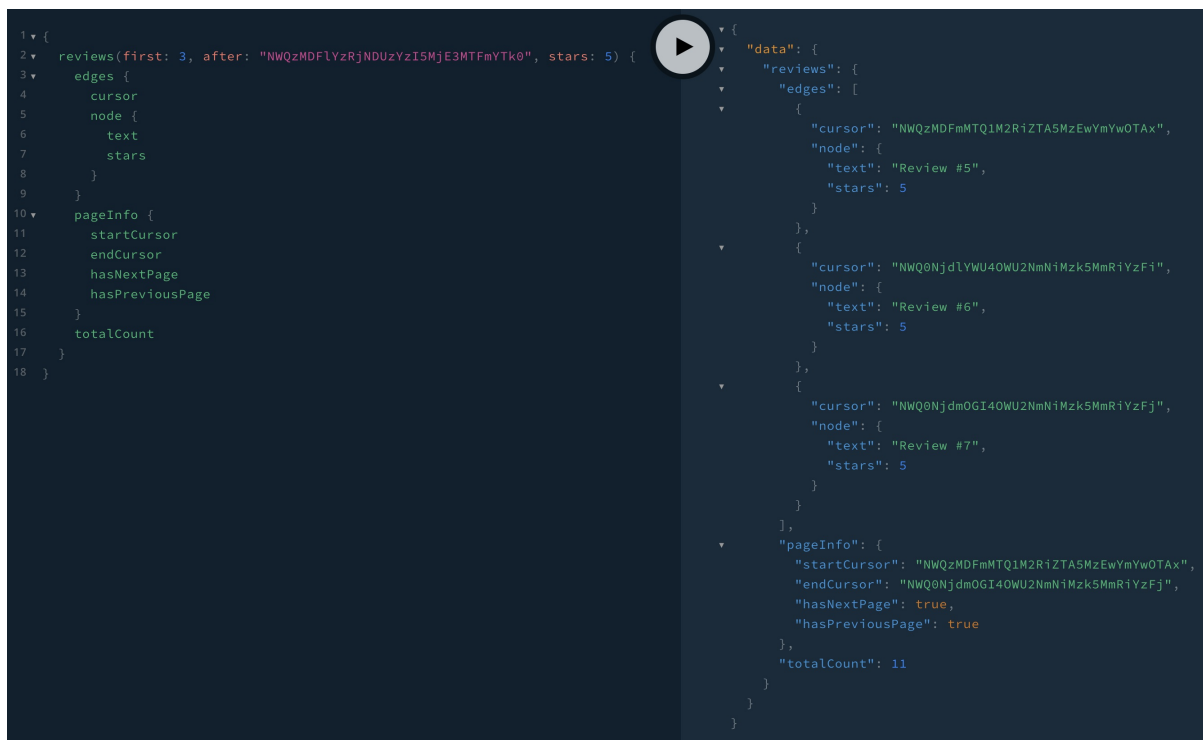
```

Apollo Server first calls the `Query.reviews` resolver, which returns a `ReviewsConnection` that includes a `PageInfo` object without a `hasPreviousPage` property. Instead, Apollo Server will call the `PageInfo.hasPreviousPage` resolver. This resolver receives as its first argument the `pageInfo` sub-object that the resolver above returned, so it can call the `getHasPreviousPage()` function, which either immediately returns a boolean (when there's no `after` argument) or initiates a database query and returns a Promise. If the `hasPreviousPage` field isn't selected in the GraphQL query, the resolver won't be called, and the database query won't be sent.

Let's try out a query:



We see there are 11 total reviews with 5 stars, starting with review #2, and there are no previous pages ( `pageInfo.hasPreviousPage` is false). If we want to request the next 3 reviews after review #4, we use `pageInfo.endCursor` as the next query's `after` :



And we get reviews #5–7 🙋🏻🤔.

Lastly, let's look at the cursor creating and decoding:

`src/util/pagination.js`

```
import { ObjectId } from 'mongodb'

export const encodeCursor = review =>
  Buffer.from(review._id.toString()).toString('base64')

export const decodeCursor = cursor =>
  ObjectId(Buffer.from(cursor, 'base64').toString('ascii'))
```

We take the review's `_id` property and base64-encode it, and then decode it back to an **ASCII** string, which we convert to an `ObjectId`.

Using `_id` works because we only support ordering by `createdAt`. If we had `orderBy: updatedAt_DESC`, then the cursor would need to contain the review's `updatedAt` property. To differentiate between the two, we could encode an object instead of just an ID string:

```
export const encodeCursor = (review, orderBy) => {
  const cursorData = ['updatedAt_DESC', 'updatedAt_ASC'].includes(orderBy)
    ? { updatedAt: review.updatedAt }
    : { _id: review._id }

  return Buffer.from(JSON.stringify(cursorData)).toString('base64')
}

export const decodeCursor = cursor =>
  JSON.parse(Buffer.from(cursor, 'base64').toString('ascii'))
```

Also, for either of our encoding systems to work, the client has to continue sending the `orderBy` and `stars` arguments (so that the server knows what MongoDB query filter and sort to use). If we wanted the client to be able to just send `first` and `after`, then we would need to encode the ordering and filtering arguments in cursors. Then the server could decode the information later when receiving a cursor as an `after` argument:

```
export const encodeCursor = (review, orderBy, stars) => {
  const cursorData = {
    _id: review._id,
    updatedAt: review.updatedAt,
    orderBy,
    stars
  }

  return Buffer.from(JSON.stringify(cursorData)).toString('base64')
}
```

## File uploads

Originally, web servers saved files to their hard drives or to colocated file servers. Most modern web servers use a third-party file-storage service like Amazon S3 or Cloudinary. When a user wants to upload a file, there are a few different ways the client can get it to a storage service:

- **Client-side:** The client sends the file directly to the storage service.
  - Signed: Our API server gives a signature to the client to give to the storage service along with the file. If our API server doesn't give the client a signature (for any reason—for example the client isn't logged in, or the logged-in user doesn't have upload permissions), then the storage service won't accept the file.
  - Unsigned: Our server is not involved, and the storage service accepts any file from any client.
- **Server-side:** The client sends the file to our server, and we forward it to the storage service.
  - Through GraphQL: The file goes through our GraphQL endpoint.
  - Outside GraphQL: We create a separate endpoint or server for the file to go through.

We recommend unsigned client-side file uploads unless the lack of signatures becomes a problem. If it does, we suggest switching to signed client-side. We prefer unsigned file uploads because they're the easiest to set up. And the client-side upload process is faster than server-side and reduces load on the GraphQL server.

Not all storage services support client-side uploads, and among those that do, only some support unsigned uploads. S3, for instance, doesn't really support it (we can configure an S3 bucket for public write access, but then anyone can delete user uploads). Cloudinary not only supports unsigned uploads, but they also take security measures to prevent abuse.

In the first section we'll go over client-side uploads, and in [the second](#) we'll do server-side through GraphQL.

### Client-side

If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...files](#))

In this section we'll add the server code to support an unsigned client-side upload—and at the end, we'll show the additional code needed for a signed upload. All we need is a mutation for the client to tell the server the filename, ID, or path, depending on which file-storage service we're using. If we wanted to make it general-purpose, we could use the file's full URL instead. For the Guide, we'll use Cloudinary, which gives the client the file's path after the upload is complete (the client-side upload process is [described in Chapter 6](#)). The server then combines the path—for example `v1551850855/jeresig.jpg`—with our account URL ( `https://res.cloudinary.com/graphql/` ) to form the full URL:

<https://res.cloudinary.com/graphql/v1551850855/jeresig.jpg>

We'll use the file-upload feature to allow users to add a profile photo (instead of using their current GitHub photo), so we'll call the mutation `setMyPhoto` and add it to `User.graphql`:

`src/schema/User.graphql`

```
extend type Mutation {
  ...
  setMyPhoto(path: String!): User!
```

```
}

```

Since `setMyPhoto` will be changing a `user` field, we return the modified `User` object.

In the resolver, we check if the client is logged in and call a new data source method `setPhoto()` :

```
src/resolvers/User.js

```

```
export default {
  ...
  Mutation: {
    createUser: ...,
    setMyPhoto(_, { path }, { user, dataSources }) {
      if (!user) {
        throw new ForbiddenError('must be logged in')
      }

      return dataSources.users.setPhoto(path)
    }
  }
}
```

The method constructs the full photo URL, saves it to the database, and returns the updated user object:

```
src/data-sources/Users.js

```

```
export default class Users extends MongoDataSource {
  ...

  async setPhoto(path) {
    const { user } = this.context
    const photo = `https://res.cloudinary.com/graphql/${path}`
    await this.collection.updateOne({ _id: user._id }, { $set: { photo } })
    return {
      ...user,
      photo
    }
  }
}
```

Now that some user documents will contain a `photo` field, we need to update our resolver:

```
src/resolvers/User.js

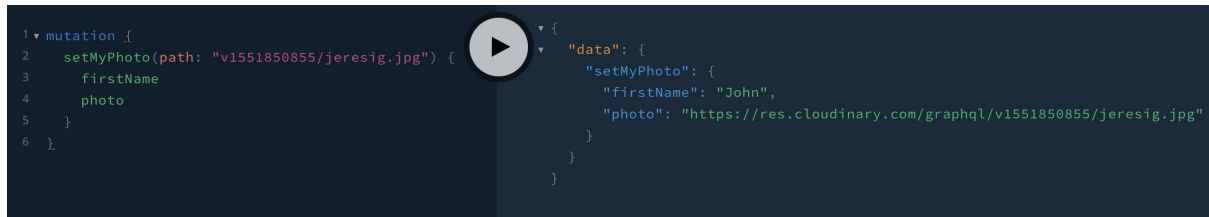
```

```
export default {
  ...
  User: {
    id: ...,
    email: ...,
    photo(user) {
      if (user.photo) {
        return user.photo
      }

      // user.authId: 'github|1615'
      const githubId = user.authId.split('|')[1]
      return `https://avatars.githubusercontent.com/u/${githubId}`
    },
    createdAt: ...
  },
  Mutation: {
    createUser: ...,
    setMyPhoto: ...
  }
}
```

We return early if the `user` object fetched from the database has a `photo` property.

We can test out the mutation in Playground with either a valid Authorization header or by hard coding the `authId` in `src/context.js`:



If we wanted to do signed client-side upload, we'd need to make a Query for the client to fetch the signature. Our resolver would call `cloudinary.utils.api_sign_request()` like this:

```
export default {
  Query: {
    ...
    uploadSignature(_, { uploadParams }, { user }) {
      if (!user) {
        throw new ForbiddenError('must be logged in')
      }

      return cloudinary.utils.api_sign_request(uploadParams, CLOUDINARY_API_SECRET)
    }
  }
}
```

Then the client would send the signature along with the file to Cloudinary's servers (and we would disable unsigned uploads in our Cloudinary account settings).

If we were using Amazon S3, then we'd use the `s3.createPresignedPost()` function to create the signature.

## Server-side

If you're jumping in here, `git checkout files_0.1.0` (tag `files_0.1.0`, or compare `files...files2`)

We go over the differences between client-side and server-side [above](#). In this section, we'll do server-side file uploads, where the client sends the file to the GraphQL server, which sends it to the storage service (we could send to Cloudinary again, but we'll use Amazon S3 this time for diversity). There are different methods for the client to send the file, and the most common is a multipart HTTP request, which works through:

- an `Upload` scalar provided by Apollo Server
- the Apollo Link `apollo-upload-client` on the client side

We create a mutation with an argument of type `Upload`:

`src/schema/User.graphql`

```
extend type Mutation {
  createUser(user: CreateUserInput!, secretKey: String!): User!
  setMyPhoto(path: String!): User!
  uploadMyPhoto(file: Upload!): User!
}
```

We'll need an instance of the AWS S3 client library ( `aws-sdk` ) to upload to S3:

`src/util/s3.js`

```
import AWS from 'aws-sdk'

export default new AWS.S3()
```

We'll import and use it in the resolver:

src/resolvers/User.js

```
import s3 from '../util/s3'

const IMAGE_MIME_TYPES = ['image/jpeg', 'image/png', 'image/gif', 'image/webp']

export default {
  ...
  Mutation: {
    ...
    uploadMyPhoto: async (_, { file }, { user, dataSources }) => {
      if (!user) {
        throw new ForbiddenError('must be logged in')
      }

      const { createReadStream, filename, mimetype } = await file

      if (!IMAGE_MIME_TYPES.includes(mimetype)) {
        throw new InputError({ file: 'must be an image file' })
      }

      const stream = createReadStream()
      const { Location: fileUrl } = await s3
        .upload({
          Bucket: 'guide-user-photos',
          Key: filename,
          Body: stream
        })
        .promise()

      return dataSources.users.setPhoto(fileUrl)
    }
  }
}
```

We first check if the user is logged in, then we check the file type (valid values taken from a [list of MIME types](#)), and then we create a Node.js file stream, which we pass to `s3.upload()` along with the filename and S3 *bucket* (the top-level folder in S3, and the subdomain of the file's URL). Finally, we call the data source `setPhoto()` method, which used to take a path, but let's refactor it to take a full URL:

src/data-sources/Users.js

```
export default class Users extends MongoDataSource {
  ...

  async setPhoto(photo) {
    const { user } = this.context
    await this.collection.updateOne({ _id: user._id }, { $set: { photo } })
    return {
      ...user,
      photo
    }
  }
}
```

Changing the parameter means we need to update where we used it previously:

src/resolvers/User.js

```
export default {
  ...
  Mutation: {
    createUser...
    setMyPhoto(_, { path }, { user, dataSources }) {
      if (!user) {
        throw new ForbiddenError('must be logged in')
      }
    }
  }
}
```

```

    return dataSources.users.setPhoto(
      `https://res.cloudinary.com/graphql/${path}`
    )
  },
  uploadMyPhoto...
}
}
}

```

We pass the full cloudinary URL instead of just the path.

In order for the AWS SDK to authenticate our account, we need to add `AWS_ACCESS_KEY_ID` and

`AWS_SECRET_ACCESS_KEY` to our `.env`.

To test this section yourself, you need an AWS account, a bucket created in the [S3 management console](#), and access keys created in the [Identity and Access Management console](#). You'd replace `'guide-user-photos'` in `src/resolvers/User.js` with your bucket name, and you'd put your own access keys in `.env`. Then you'd write [a test like this](#) or create a small web app that used `apollo-upload-client` to send a file in an `uploadMyPhoto` Mutation.

When the `uploadMyPhoto` Mutation is run, the upload is successful, and the server saves a URL like this in the `photo` field of the current user's MongoDB document:

```
https://guide-user-photos.s3.amazonaws.com/filename.jpg
```

## Schema validation

In this section we'll go over schema validation and how to set it up using [Apollo Graph Manager](#).

There are three places where our server is currently doing things that we might call schema validation:

- `gql` parses our [SDL](#) strings and throws errors when they're invalid.
- On startup, `ApolloServer` checks the `typeDefs` it receives to see if our whole schema is valid, according to the GraphQL spec.
- While running, `ApolloServer` validates queries against the schema.

However, usually the term *schema validation* refers to schema-change validation: i.e., ascertaining whether a *change* to a schema is valid. When we deploy a schema and clients use it, and we then change the schema and want to re-deploy, we can first use schema validation to check if the change is valid. "Valid" in this context can have different meanings. We could say it's invalid if any of the changes are backward incompatible. However, sometimes we want to make backward-incompatible changes. So, often "valid" means the changes will work with X% of queries in the last N days. The default for Apollo Graph Manager is 100% of queries in the last seven days. This way, backward-incompatible changes can be made as long as no clients have selected the changed field within the past week.

`graphql-inspector` is a command-line tool for [finding breaking or dangerous changes](#), and [GraphQL Doctor](#) is a GitHub app that does the same for pull requests, comparing the PR's schema against the schema in `master`. However, we recommend using Graph Manager if you can (the validation feature requires a paid plan). Its method of validating against the query patterns of our clients is more broadly useful, and it's easy to use from the command line, in continuous integration, and in GitHub PRs.

The first step to setting up Graph Manager is setting the env var `ENGINE_API_KEY` to the value we get from our [Graph Manager account](#). We already added it to our `.env` in the [Analytics](#) section. Having `ENGINE_API_KEY` configures the `apollo` command-line tool, which we use for schema registration and validation, and it enables metrics reporting (which we need for validation, because validation is based on clients' queries, which are collected metrics).

The second step we also did in the Analytics section: Registering our schema with Graph Manager. Let's assume we have our app running in production at `api.graphql.guide`. We would register the production schema with:



```
$ npx apollo service:push --endpoint="https://api.graphql.guide/graphql" --tag=prod
```

We use `--tag` to denote the *variant*. Graph Manager tracks variants of schemas, each with their own metrics and schema history. So the above command says to Apollo: “Introspect the schema at `api.graphql.guide` and save it as the latest version of our ‘prod’ schema variant.”

Registration has other uses beyond validation—it also powers the [Apollo VS Code extension](#) and Graph Manager’s schema history and analytics.

Then, when we make changes to our schema, before we push to production, we check to see whether the change is valid by running `npm run dev` in one terminal and the following in another:

```
$ npx apollo service:check --endpoint="http://localhost:4000/graphql" --tag=prod
```

This says, “Introspect the schema of the server running on port 4000 of my machine and validate it against the latest production schema.” It will output either success or a list of which changes fail validation, like this:

```
$ npx apollo service:check ...
✓ Loading Apollo Project
✓ Validated local schema against tag prod on service engine
✓ Compared 8 schema changes against 110 operations over the last 7 days
✗ Found 2 breaking changes and 3 compatible changes
  → breaking changes found

FAIL    ARG_REMOVED      `Query.searchUsers` arg `term` was removed
FAIL    FIELD_REMOVED     `Review.stars` was removed

PASS    FIELD_ADDED        `Review.starCount` was added
PASS    ARG_ADDED          `Query.searchUsers` arg `partialName` was added
PASS    TYPE_REMOVED       `ReviewComment` removed
PASS    FIELD_DEPRECATION_REMOVED `Review.text` is no longer deprecated

View full details at: https://engine.apollographql.com/service/example-123/check/foo
```

Given the validation failure, we would know to not push to production.

We can save ourselves time and the risk of forgetting to run the validation command by automating it—for instance, with the [Apollo Engine GitHub App](#) or with a continuous integration service like CircleCI:

```
.circleci/config.yml
```

```
version: 2

jobs:
  validate_against_production:
    docker:
      - image: circleci/node:8

    steps:
      - checkout

      - run: npm install

      - run:
          name: Starting server
          command: npm start
          background: true

      # Wait for server to start up
      - run: sleep 5

      - run: npx apollo service:check --endpoint="http://localhost/graphql" --serviceName=users --tag=prod
```

Validating Apollo federation services is similar, and we’ll see how in the [Managed federation](#) section below.

## Apollo federation

- [Federated service](#)
- [Federated gateway](#)
- [Extending entities](#)
- [Managed federation](#)
- [Deploying federation](#)

In the [Introduction](#) to this chapter, we talk about microservices versus monoliths. If we go down the microservice route, then the best way to do it is with Apollo federation.

Apollo federation is a specification for how to divide our schema across different services. Each service describes which parts of the schema it implements, and a gateway combines all the parts into one larger schema. The gateway stands between the client and the services, receiving requests from the client and automatically resolving them through one or more requests to services.

The Apollo federation specification can be implemented in any language and has been added to many [existing GraphQL server libraries](#). Those servers that follow the specification are the services, and the gateway is a special instance of Apollo Server that uses the `@apollo/gateway` library.

In the first three sections, we'll rebuild our Guide server monolith using federation: We'll start with a users service, then the gateway, and then the reviews service. Then in [Managed federation](#), we'll see how we can benefit from Apollo's Graph Manager SaaS product, and finally in [Deploying federation](#), we'll discuss the deployment of the gateway and services.

### Federated service

If you're jumping in here, `git checkout federation_0.1.0` (tag [federation\\_0.1.0](#), or compare [federation...federation2](#))

In this section we'll build a users service: A GraphQL server that supports Apollo federation and handles queries related to the `User` type. We'll start from a new tag:

```
$ git checkout federation_0.1.0
```

Here is our starting file structure:

```
$ tree -L 3
.
├── babel.config.json
├── lerna.json
├── lib
│   ├── Date.js
│   ├── auth.js
│   ├── db.js
│   └── errors.js
├── package.json
└── services
    ├── reviews
    │   └── package.json
    └── users
        └── package.json
```

The two services will go in the `services/` folder, and `lib/` contains code to share between the services (taken from the monolith we built earlier). Let's install all the modules we need:

```
$ npm install
```

This creates a `node_modules/` at the root—which has modules for the gateway code that we'll place at the root—and it also creates `node_modules/` folders inside `services/reviews/` and `services/users/` thanks to the [Lerna library](#), which we configure in `lerna.json` and use in a `postinstall` script in `package.json`:

```
{
  "name": "guide-api",
  "version": "0.1.0",
  "scripts": {
    "start": "babel-watch gateway.js",
    "start-service-users": "babel-watch services/users/index.js",
    "start-service-reviews": "babel-watch services/reviews/index.js",
    "start-services": "concurrently \"npm:start-service-*\"",
    "postinstall": "lerna bootstrap"
  },
  ...
}
```

We also see from the scripts where we'll locate the main server files:

```
gateway.js
services/users/index.js
services/reviews/index.js
```

`concurrently` runs multiple other scripts in the same terminal—in this case, both `start-service-users` and `start-service-reviews`.

In this section, we'll be filling in `services/users/*`. There are three main parts to a federated service:

- `buildFederatedSchema()`: Instead of passing `typeDefs` and `resolvers` directly to `ApolloServer()`, we give them to the `buildFederatedSchema()` from the `@apollo/federation` library.
- *Entities*: Types defined in one service that can be referenced or extended by other services.
  - `@key` directive: Each entity requires a `@key` directive denoting the primary key.
  - `__resolveReference()`: For each entity, we must write a reference resolver, which fetches an entity object by its `@key` field(s).

As usual, let's start with the schema:

`services/users/schema.js`

```
import { gql } from 'apollo-server'

export default gql`
  scalar Date

  extend type Query {
    me: User
    user(id: ID!): User
  }

  type User @key(fields: "id") {
    id: ID!
    firstName: String!
    lastName: String!
    username: String!
    email: String
    photo: String!
    createdAt: Date!
    updatedAt: Date!
  }
`
```

We include shared types like custom scalars in the schema of each service. Also, the `Query` and `Mutation` types will be initially defined in the gateway, so the services `extend` them. Finally, our `User` type has this directive:

`@key(fields: "id")`, which tells the gateway that the `User` type is a federation entity and the `id` field is its primary key.

We copy the below from our monolith's `src/resolvers/User.js` with a couple of additions:

- Adding the `Date` resolvers, imported from `lib/Date.js`
- Adding `User.__resolveReference`

`services/users/resolvers.js`

```
import { ForbiddenError } from 'apollo-server'
import { ObjectId } from 'mongodb'

import { InputError } from '../../lib/errors'
import Date from '../../lib/Date'

const OBJECT_ID_ERROR =
  'Argument passed in must be a single String of 12 bytes or a string of 24 hex characters'

export default {
  ...Date,
  Query: {
    me: (_, __, context) => context.user,
    user: (_, { id }, { dataSources }) => {
      try {
        return dataSources.users.findOneById(ObjectId(id))
      } catch (error) {
        if (error.message === OBJECT_ID_ERROR) {
          throw new InputError({ id: 'not a valid Mongo ObjectId' })
        } else {
          throw error
        }
      }
    },
  },
  User: {
    __resolveReference: (reference, { dataSources }) =>
      dataSources.users.findOneById(ObjectId(reference.id)),
    id: ({ _id }) => _id,
    email(user, _, { user: currentUser }) {
      if (!currentUser || !user._id.equals(currentUser._id)) {
        throw new ForbiddenError('cannot access others' emails')
      }

      return user.email
    },
    photo(user) {
      // user.authId: 'github|1615'
      const githubId = user.authId.split('|')[1]
      return `https://avatars.githubusercontent.com/u/${githubId}`
    },
    createdAt: user => user._id.getTimestamp()
  }
}
```

The first argument to `__resolveReference` is the reference: An object containing the primary key field(s)—in this case, just the `id`—which we resolve to the user object.

Now we put the resolvers and schema together to create the server:

`services/users/index.js`

```
import { ApolloServer } from 'apollo-server'
import { buildFederatedSchema } from '@apollo/federation'
import { MongoDataSource } from 'apollo-datasource-mongodb'

import resolvers from './resolvers'
```

```
import typeDefs from '../schema'
import { MongoClient } from '../../lib/db'
import context from '../../lib/userContext'

const server = new ApolloServer({
  schema: buildFederatedSchema([
    {
      typeDefs,
      resolvers
    }
  ]),
  dataSources: () => ({
    users: new MongoDataSource(mongoClient.db().collection('users'))
  }),
  context
})

mongoClient.connect()

server.listen({ port: 4001 }).then(({ url }) => {
  console.log(`Users service ready at ${url}`)
})
```

Here we see the use of `buildFederatedSchema()`. Also, the only data source method we use is `.findOneById()`, so we can use `MongoDataSource` directly instead of defining a subclass. `MongoClient` we get from `db.js`:

`lib/db.js`

```
import { MongoClient } from 'mongodb'

const URL = 'mongodb://localhost:27017/guide'

export const mongoClient = new MongoClient(URL)
```

Finally, our `context` function needs to provide a `user` object for the `Query.me` resolver. Our monolith context function looked at the `authorization` header, decoded the `authId`, and fetched the user object. Instead of having each of our services repeat this process, we can have our gateway do part or all of it. We can either do:

1. Gateway decodes `authId` and passes it to services as an `auth-id` header. Services read the header and fetch the user document.
2. Gateway decodes `authId`, connects to the user database to fetch the user document, and passes it to services as a `user` header.
3. The JWT that's sent in the authorization header from the client can be created to contain the whole user document, so that when it's decoded, no database query is required.

Our JWTs don't have the whole user document, so we can't do #3. Between #1 and #2, #2 is more efficient, as it reduces the number of database calls. Note that #2 isn't possible when the user document is large. The maximum header size is set by the receiving server, for instance Nginx has a maximum 4KB, which is ~4,000 ASCII characters. (We can check the length of a user document by doing `JSON.stringify(user).length`.) Here is the service side of #2:

`lib/userContext.js`

```
module.exports = async ({ req }) => {
  const context = {}

  const userDocString = req && req.headers['user']
  if (userDocString) {
    context.user = JSON.parse(userDocString)
  }

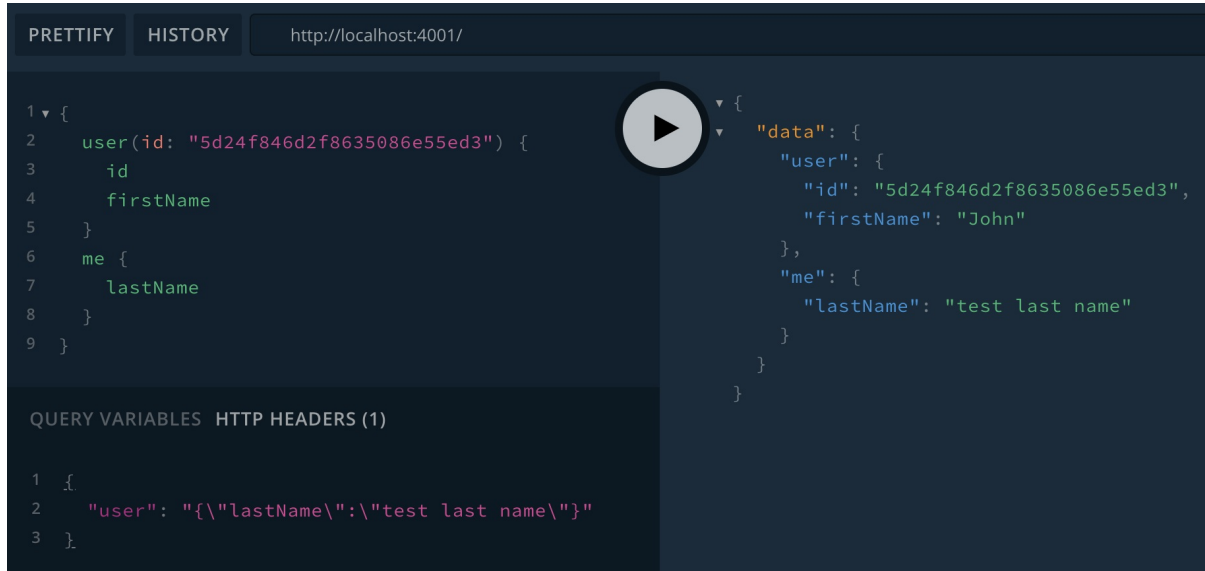
  return context
}
```

Now we can set the `user` HTTP header and both `Query.user` and `Query.me` work:

```
$ npm run start-service-users

> guide-api@0.1.0 start-service-users /guide-api
> babel-watch services/users/index.js

Users service ready at http://localhost:4001/
```



## Federated gateway

If you're jumping in here, `git checkout federation2_0.1.0` (tag [federation2\\_0.1.0](#), or compare [federation2...federation3](#))

In the last section we implemented the users service. In this section, we'll implement the gateway. The basic process is creating an `ApolloGateway()` that points to a list of the services, and then giving that to `ApolloServer()`:

`gateway.js`

```
import { ApolloServer } from 'apollo-server'
import { ApolloGateway } from '@apollo/gateway'

const gateway = new ApolloGateway({
  serviceList: [
    { name: 'users', url: 'http://localhost:4001/graphql' },
  ]
})

const server = new ApolloServer({
  gateway,
  subscriptions: false
})

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

We disable subscriptions because they don't yet work with `ApolloGateway`. This works, but it's not yet sending the `user` HTTP header our `users` service expects. This takes two steps: copying our monolith's context function to give to `ApolloServer()` and defining a `buildService()` function to add the header in requests to services:

```
import { ApolloServer } from 'apollo-server'
import { ApolloGateway, RemoteGraphQLDataSource } from '@apollo/gateway'

import context from './context'
```

```
import { MongoClient } from './lib/db'

class AuthenticatedDataSource extends RemoteGraphQLDataSource {
  willSendRequest({ request, context }) {
    request.http.headers.set('user', JSON.stringify(context && context.user))
  }
}

const gateway = new ApolloGateway({
  serviceList: [
    { name: 'users', url: 'http://localhost:4001/graphql' },
    { name: 'reviews', url: 'http://localhost:4002/graphql' }
  ],
  buildService({ url }) {
    return new AuthenticatedDataSource({ url })
  }
})

const server = new ApolloServer({
  gateway,
  context,
  subscriptions: false
})

mongoClient.connect()

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

`buildService()` returns an `AuthenticatedDataSource` which sets the stringified user doc from the context as a header. `willSendRequest()` is then called for each request from the gateway to the services. We also import `MongoClient` in order to initiate the connection and import context from:

`context.js`

```
import { AuthenticationError } from 'apollo-server'

import { getAuthIdFromJWT } from './lib/auth'
import { MongoClient } from './lib/db'

export default async ({ req }) => {
  const context = {}

  const jwt = req && req.headers.authorization
  let authId

  if (jwt) {
    try {
      authId = await getAuthIdFromJWT(jwt)
    } catch (e) {
      let message
      if (e.message.includes('jwt expired')) {
        message = 'jwt expired'
      } else {
        message = 'malformed jwt in authorization header'
      }
      throw new AuthenticationError(message)
    }
  }

  const user = await mongoClient
    .db()
    .collection('users')
    .findOne({ authId })
  if (user) {
    context.user = user
  } else {
    throw new AuthenticationError('no such user')
  }
}

return context
```

```
}
```

The only difference between this and the monolith's version is importing `mongoClient` instead of the `db` directly.

We can now run our `users` service and gateway in two different terminals:

```
$ npm run start-service-users

> guide-api@0.1.0 start-service-users /guide-api
> babel-watch services/users/index.js

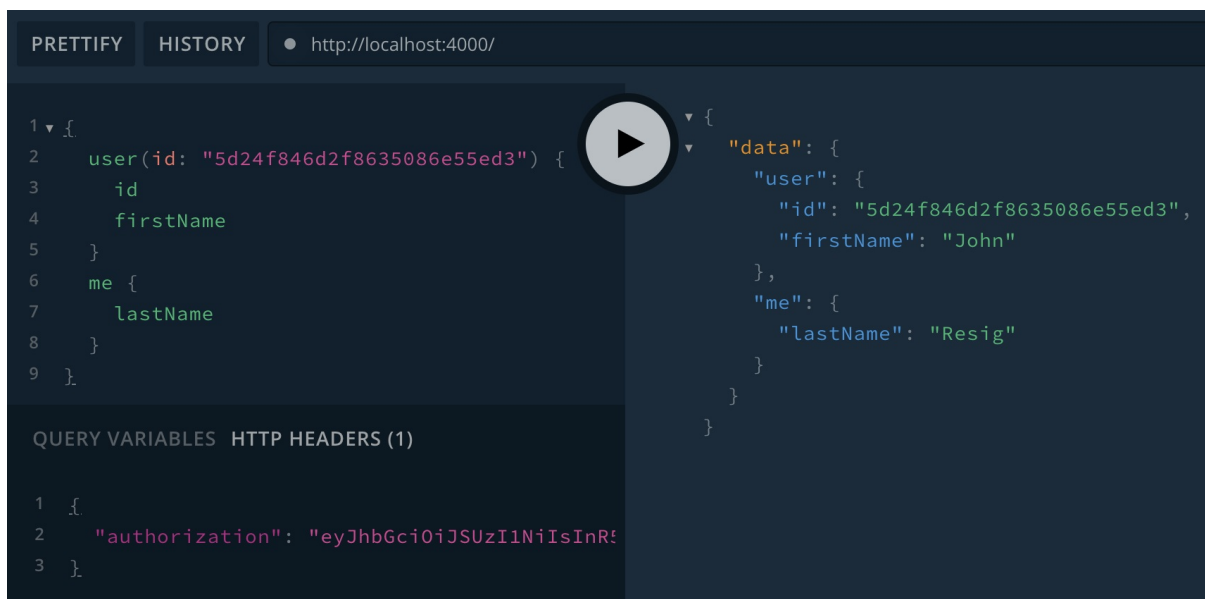
Users service ready at http://localhost:4001/
```

```
$ npm start

> guide-api@0.1.0 start /guide-api
> babel-watch gateway.js

Gateway ready at http://localhost:4000/
[INFO] Wed Mar 1 2020 04:55:43 GMT-0400 (EST) apollo-gateway: Gateway successfully loaded schema.
* Mode: unmanaged
```

When we open the gateway URL, set our authorization header, and query, it works! 🎉



## Extending entities

If you're jumping in here, `git checkout federation3_0.1.0` (tag [federation3\\_0.1.0](#), or compare [federation3...federation4](#))

In this section, we'll build another service—this one for reviews—and we'll see how to extend entities created by other services. Then, we'll add the reviews service to the gateway and see how the gateway resolves queries involving both services.

Let's start with the schema. First, we take the `Review` type and `reviews` query from our monolith for our new schema, and then we add a few things:

```
services/reviews/schema.js
```

```
import { gql } from 'apollo-server'
```



```
export default gql`
  scalar Date

  type Review @key(fields: "id") {
    id: ID!
    text: String!
    stars: Int
    author: User!
    createdAt: Date!
    updatedAt: Date!
  }

  extend type Query {
    reviews: [Review!]!
  }

  extend type User @key(fields: "id") {
    id: ID! @external
    reviews: [Review!]!
  }
`
```

- `scalar Date`, as we did in the `users` service
- `@key` directive for `type Review`, to declare it as a federation entity
- `extend type User`: Here we're extending the `User` type originally defined externally. We have to include both the `@key` directive as well as the primary key fields—in this case just `User.id`—with the `@external` directive (signifying that this field was originally defined in another service). The `reviews` field doesn't have `@external`, which means it's being added to the `User` type, and we'll need to write a resolver for it:

`services/reviews/resolvers.js`

```
import { ObjectId } from 'mongodb'

import Date from '../lib/Date'

export default {
  ...Date,
  Query: {
    reviews: (_, __, { dataSources }) => dataSources.reviews.all(),
  },
  Review: {
    __resolveReference: (reference, { dataSources }) =>
      dataSources.reviews.findOneById(ObjectId(reference.id)),
    id: review => review._id,
    author: review => ({ id: review.authorId }),
    createdAt: review => review._id.getTimestamp(),
  },
  User: {
    reviews: (user, __, { dataSources }) =>
      dataSources.reviews.all({ authorId: ObjectId(user.id) })
  }
}
```

These resolvers are taken from our monolith with four additions:

- The `Date` custom scalar resolver.
- The `Review.__resolveReference` resolver, required because this service is the origin of the `Review` entity.
- The `Review.author` resolver, which returns a `reference` (the same reference passed to `__resolveReference` above)—an object with an entity's primary key. The gateway takes this reference and provides it to the `User.__resolveReference` resolver to get the user object.
- The `User.reviews` resolver, which uses the data source `review.all()` method with a MongoDB selector. Speaking of which, we need a `Reviews` data source with a `.all()` method:

`services/reviews/Reviews.js`

```
import { MongoDataSource } from 'apollo-datasource-mongodb'
```

```
export default class Reviews extends MongoDataSource {
  all(query) {
    return this.collection.find(query).toArray()
  }
}
```

We'll include this, along with our schema and resolvers, when creating the server:

`services/reviews/index.js`

```
import { ApolloServer } from 'apollo-server'
import { buildFederatedSchema } from '@apollo/federation'

import resolvers from './resolvers'
import typeDefs from './schema'
import Reviews from './Reviews'
import { MongoClient } from 'mongoose'
import context from '../lib/userContext'

const server = new ApolloServer({
  schema: buildFederatedSchema([
    {
      typeDefs,
      resolvers
    }
  ]),
  dataSources: () => ({
    reviews: new Reviews(MongoClient.db().collection('reviews'))
  }),
  context
})

MongoClient.connect()

server.listen({ port: 4002 }).then(({ url }) => {
  console.log(`Reviews service ready at ${url}`)
})
```

We use the same context function as the `users` service and a new port (4002, versus 4001 for the `users` service and the default 4000 for the gateway).

One piece of our old schema that we're missing is `Review.fullReview`. Since it involves the author's name, we need to query the `users` collection. And the service that is responsible for querying the `users` collection is the `users` service. So let's add the field to the `users` service:

`services/users/schema.js`

```
export default gql`
  ...

  extend type Review @key(fields: "id") {
    id: ID! @external
    fullReview: String!
  }
`
```

Like with `extend type User`, when we `extend type Review`, we repeat the directive and include the primary key field. However, we have an issue: The `fullReview` resolver needs data from the review document (`authorId`, `text`, and `stars`). By default, the resolver will only receive an object with the review's `id` field.

We can solve this issue with the `@requires` directive:

```
export default gql`
  ...
```

```

extend type Review @key(fields: "id") {
  id: ID! @external
  text: String! @external
  stars: Int @external
  authorId: ID! @external
  fullReview: String! @requires(fields: "authorId text stars")
}

```

We list the fields we require in order to resolve `fullReview` using `@requires`, and we list those fields above with `@external`. The last issue is that `authorId` isn't currently part of the `Review` type, so let's add it to the `reviews` service schema:

`services/reviews/schema.js`

```

export default gql`
  scalar Date

  type Review @key(fields: "id") {
    id: ID!
    text: String!
    stars: Int
    authorId: ID!
    author: User!
    createdAt: Date!
    updatedAt: Date!
  }

  ...
`

```

This makes `authorId` appear in the public gateway schema as well, which isn't ideal, as it unnecessarily clutters the schema, but the ability to define a private, internal field is a [planned addition](#) to the federation spec.

Finally, we can implement the `fullReview` resolver back in the `users` service:

`services/users/resolvers.js`

```

export default {
  ...
  Review: {
    fullReview: async (review, _, { dataSources }) => {
      const author = await dataSources.users.findOneById(
        ObjectId(review.authorId)
      )
      return `${author.firstName} ${author.lastName} gave ${review.stars} stars, saying: "${review.text}"`
    }
  }
}

```

We add the `reviews` service to our gateway by simply adding it to our `serviceList`:

`gateway.js`

```

const gateway = new ApolloGateway({
  serviceList: [
    { name: 'users', url: 'http://localhost:4001/graphql' },
    { name: 'reviews', url: 'http://localhost:4002/graphql' }
  ],
  buildService({ url }) {
    return new AuthenticatedDataSource({ url })
  }
})

```

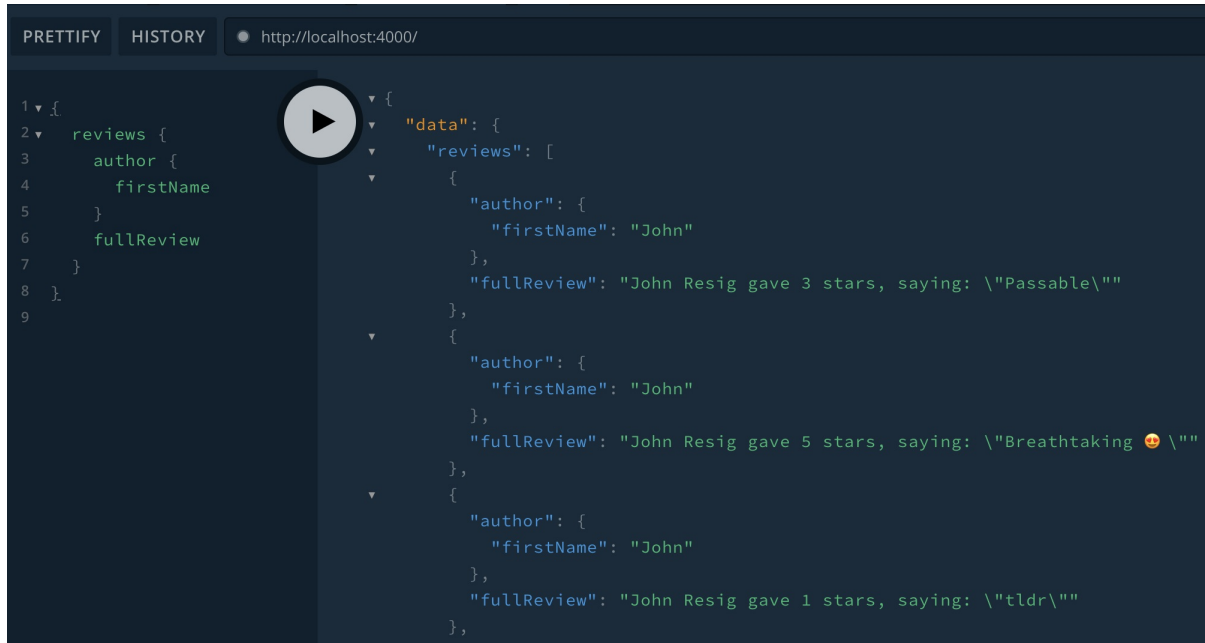
We can run both services with:

```
$ npm run start-services
```

And in another terminal run the gateway:

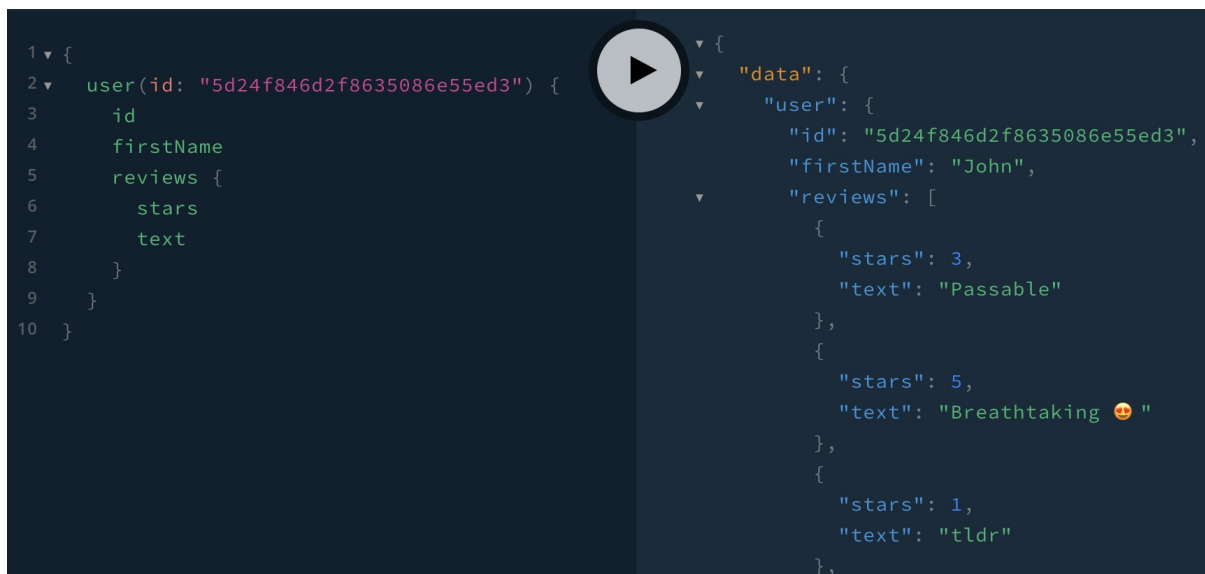
```
$ npm start
```

And test! 🙏



✓ Here we see both of the jumps from the `reviews` service to the `users` service working: The `reviews` service resolves `Query.reviews` and the `Review.author` reference, and the `users` service resolves the reference into a user, as well as `User.firstName` and `Review.fullReview`.

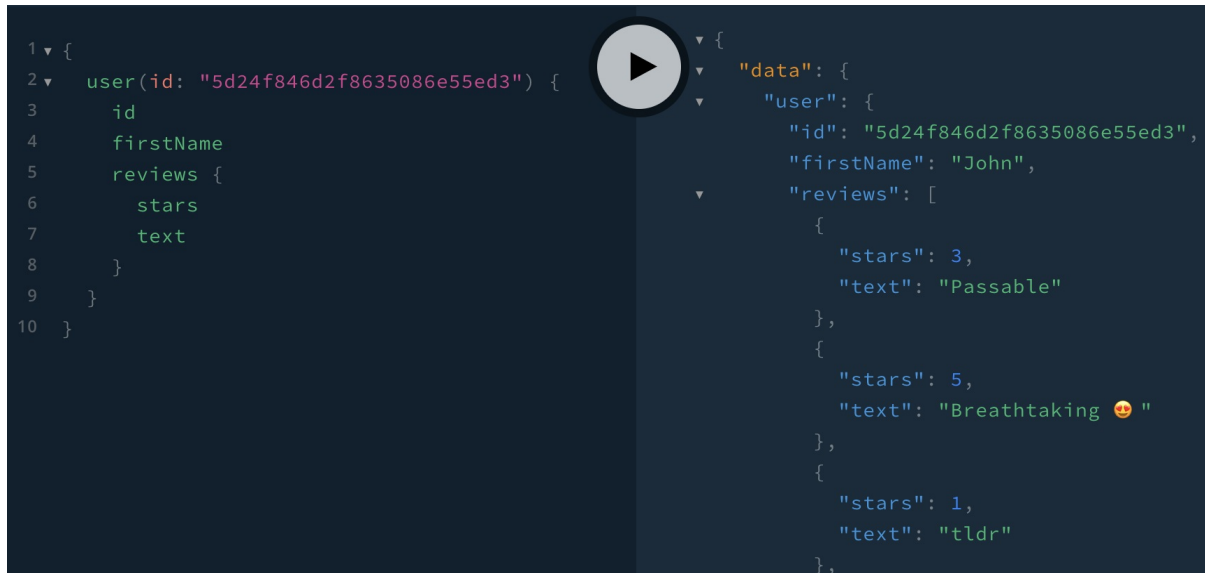
Next, we can see that going from the `users` service to the `reviews` service works. First the `users` service resolves `query.user`, and then the `reviews` service resolves `User.reviews`.



To see a more detailed explanation of the *query plan*—the process by which the gateway determines how to get all the data it needs from the services—we can add this last argument to `ApolloGateway()`:

```
const gateway = new ApolloGateway({
  serviceList...
  buildService...
  __exposeQueryPlanExperimental: true
})
```

Now inside Playground, we can open the QUERY PLAN tab on the bottom-right:



```
{
  user(id: "5d24f846d2f8635086e55ed3") {
    id
    firstName
    reviews {
      stars
      text
    }
  }
}
```

The above query results in the below query plan:

```
QueryPlan {
  Sequence {
    Fetch(service: "users") {
      {
        user(id: "5d24f846d2f8635086e55ed3") {
          id
          firstName
          __typename
        }
      }
    },
    Flatten(path: "user") {
      Fetch(service: "reviews") {
        {
          ... on User {
            __typename
            id
          }
        } =>
        {
          ... on User {
            reviews {
              stars
              text
            }
          }
        }
      }
    }
  }
}
```

```

    },
  },
},
}

```

`Sequence` means the following queries are done in sequence—one after the other. So first it does a `Fetch` from the `users` service, and then a fetch from the `reviews` service.

Our first query involves a `Parallel` in addition to a `Sequence` :

```

{
  reviews {
    author {
      firstName
    }
    fullReview
  }
}

```

```

QueryPlan {
  Sequence {
    Fetch(service: "reviews") {
      {
        reviews {
          author {
            __typename
            id
          }
          __typename
          id
          authorId
          text
          stars
        }
      }
    },
    Parallel {
      Flatten(path: "reviews.@") {
        Fetch(service: "users") {
          {
            ... on Review {
              __typename
              id
              authorId
              text
              stars
            }
          } =>
          {
            ... on Review {
              fullReview
            }
          }
        },
      },
      Flatten(path: "reviews.@.author") {
        Fetch(service: "users") {
          {
            ... on User {
              __typename
              id
            }
          } =>
          {
            ... on User {
              firstName
            }
          }
        },
      },
    },
  },
}

```

```

    },
  },
},
}

```

The gateway first fetches from the `reviews` service and then does two fetches from the `users` service for each review, all in parallel.

We can look at the query plan to diagnose performance issues—it's possible that the query plan will show a lot of fetches in series, which increases latency. A fetch in series—where the second fetch happens after the first is complete—is denoted by `Sequence`. In the case of bugs, the query plan might also help us discover why the gateway is not working as we expect.

Another tool we have for diagnosing bugs is our gateway's `RemoteGraphQLDataSource`, to which we can add the `didReceiveResponse` method, where we can log responses from the services:

```

class AuthenticatedDataSource extends RemoteGraphQLDataSource {
  willSendRequest...

  didReceiveResponse({ response, request, context }) {
    console.log('response data:', response.data)
    return response
  }
}

```

Here are further capabilities we aren't using:

- Having [multiple primary keys](#) or [compound primary keys](#)
- Resolving other services' fields with the `@provides` directive
- [Modifying the gateway's response](#)
- Using [custom directives](#)

## Managed federation

As we've been running the gateway, we've been seeing the output:

```
* Mode: unmanaged
```

The default gateway mode is unmanaged. A gateway is *managed* when it's connected to Apollo Graph Manager, the SaaS tool we've used previously for [Analytics](#) and [Schema validation](#). `ApolloGateway` will connect to Graph Manager if we set `ENGINE_API_KEY` and make one change to the code—remove the `serviceList` argument in the constructor:

gateway.js

```

const gateway = new ApolloGateway({
  serviceList: [
    { name: 'users', url: 'http://localhost:4001/graphql' },
    { name: 'reviews', url: 'http://localhost:4002/graphql' }
  ],
  buildService({ url }) {
    return new AuthenticatedDataSource({ url })
  },
  __exposeQueryPlanExperimental: true
})

```

In managed federation, instead of listing the service URLs in the gateway, we register each service with Graph Manager, and the gateway gets the service info from Graph Manager. This has two main benefits:

1. When we add services, change service URLs, or change service schemas, we don't need to redeploy the

gateway.

2. When there's an error with one of the changes in #1, the gateway can automatically fall back to the last working configuration.

We register a service in the same way we registered our monolith's schema in [Analytics](#) and [Schema validation](#)—with the `apollo service:push` command:

```
$ npx apollo service:push \
  --serviceName=users \
  --serviceURL="http://users.svc.cluster.local:4001/" \
  --endpoint="http://localhost:4001/"
```

We can view the list of services we've pushed:

```
$ npx apollo service:list
✓ Loading Apollo Project
✓ Fetching list of services for graph guide-api
```

name	URL	last updated
Users	http://users.svc.cluster.local:4001/	5 March 2020 (5 days ago)
Reviews	http://reviews.svc.cluster.local:4002/	5 March 2020 (5 days ago)

View full details at: <https://engine.apollographql.com/graph/guide-api/service-list>

To validate the service, we use `--serviceName` with the `apollo service:check` command we used in the [Schema validation](#) section:

```
$ npx apollo service:check \
  --serviceName=users \
  --endpoint="http://localhost:4001/" \
  --tag=prod \
```

Just as monolith schemas can have multiple *variants*, denoted by the `--tag` option, so can federated schemas.

This command not only validates the service's schema against recent usage data, but it also checks failed composition—that is, a failure in the ability to compose the whole federated schema.

Now we know how to set up Graph Manager with federation and to validate changes to services to make sure they continue to fit into the whole data graph and don't break clients.

## Deploying federation

The gateway and our services are all just Node.js servers, so we can use any of the deployment options we discussed in the main [Deployment section](#). And Apollo gateway doesn't yet support subscriptions, so FaaS websocket support isn't an issue like it was before. One new issue is the recommendation that services not be publicly accessible. Federation services need to expose extra information to work with the gateway (note the added `_service` and `_entities` root query fields), and we might not want people to be able to access it.

There are a number of different options for deploying services privately, including:

- IaaS or Faas: Amazon's VPC ([Virtual Private Cloud](#)) with either EC2 or Lambda
- PaaS: Heroku's [Private Spaces](#) (requires an Enterprise account)
- Kubernetes [private clusters](#)

And if we didn't care about the information exposure, we could use public-only options like Vercel Now.

There are three steps we usually do around deployment:



- Schema validation ( `apollo service:check` )
- Code deployment (various)
- Push new service information to Graph Manager ( `apollo service:push` )

Normally, it's best to do them in the order listed—first checking if the service's schema will fit in the graph and not break queries, then deploying the code, and finally, once the production servers are ready to receive requests, telling the gateway about the updated service. In CircleCI, it would look something like this:

`.circleci/config.yml`

```
version: 2

jobs:
  deploy_to_prod:
    docker:
      - image: circleci/node:8

    steps:
      - checkout

      - run: npm install

      - run:
          name: Starting server
          command: npm start
          background: true

      # Wait for server to start up
      - run: sleep 5

      - run: npx apollo service:check --serviceName=users --endpoint="http://localhost/graphql" --tag=prod

      - run: npm run deploy

      # Wait for production servers to restart
      - run: sleep 5

      - run: npx apollo service:push --serviceName=users --endpoint="http://localhost/graphql/" --tag=prod
```

If the `service:check` command fails, the CircleCI build will fail, and `npm run deploy` and subsequent commands won't get run.

When a `service:push` is not backward compatible with our gateway's query planner (for instance when we change `@key @requires @provides` directives), then we should do the `service:push` *before* deploying. And generally, when we make modifications that affect the query planner, we need to take the steps listed in [Apollo Docs: Modifying query-planning logic](#). The article has different instructions for *in-place* versus *atomic* changes. In-place is when we deploy a service to the same domain, whereas atomic is when we deploy a service to a new domain and `service:push` to point the gateway at the new domain. Let's look at the difference using Vercel Now, which creates a unique URL with every deployment.

In-place, deploying to the existing `serviceUrl` :

```
$ apollo service:push \
  --tag=prod
  --serviceName=users
  --endpoint="http://localhost:4001"
$ now --prod
> https://users.api.graphql.guide
> Success! Deployment ready
```

Atomic, changing the `serviceUrl` :

```
$ now
> https://users-61h1hvwis.now.sh/
```

```
> Success! Deployment ready
$ apollo service:push \
  --tag=prod \
  --serviceName=users \
  --endpoint="http://localhost:4001" \
  --serviceUrl="https://users-61h1hvwis.now.sh/"
```

In summary, we started out this Apollo federation section by building a [users service](#) and connecting it to a [gateway](#). Then we built a [second service](#) for reviews and extended entities. Finally, we learned how to set up [managed federation](#) and [how to deploy](#).

## Hasura

Background: [Databases](#), [SQL](#)

[Hasura](#) is a GraphQL BaaS ([Backend as a Service](#)). In [Deployment > Options](#) we covered IaaS, PaaS, and FaaS, which are different ways we can host our code. In BaaS, we don't have to write code—the server and database (PostgreSQL in the case of Hasura) are automatically set up based on our configuration.

While it's true we don't *have* to write code, many apps need at least a little custom logic, so there are various ways to write our own code or SQL statements and integrate them into our Hasura server's functioning. These ways—which we'll get to later in this section—include actions, triggers, functions, and remote schemas.

*[Note: the rest of this section is forthcoming]*

## Schema design

- [One schema](#)
- [User-centric](#)
- [Easy to understand](#)
- [Easy to use](#)
- [Mutations](#)
  - [Arguments](#)
  - [Payloads](#)
- [Versioning](#)

### One schema

Ash graph durbatulûk, ash graph gimbatul, ash graph thrakatulûk, agh gateway-ishi krimpatul.

Inscription upon the Ring of Byron, written in Black Speech. Translates as:

One graph to rule them all, one graph to find them, one graph to bring them all, and in the gateway bind them.

The first principle of schema design is there should only be one schema! While we can *implement* it as smaller schemas and a [federation gateway](#), from the perspective of the client, there should only be one schema (or *data graph*). And while this may seem obvious, there are many large companies whose GraphQL adoption began by independent teams creating their own GraphQL APIs. This results in a lot of duplication of effort—not only duplicated resolvers where the schemas overlap, but also management of the APIs. We also might wind up with clients that need to make requests from two separate endpoints, which our frontend devs might find... inconvenient 😊. Which brings us to the first principle of design in general, which is:

## User-centric

### Design things for the people who will be using them.

The people who will be using our schema are primarily our frontend devs (or, in the case of a public API, the world's frontend devs 😊), so we want to design the schema for them. We want our API to be:

- Easy to understand.
- Easy to use.
- Hard for devs to make mistakes or create bugs when querying.

Secondarily, our schema is used by our end users (the people using the software written by the frontend devs) and ourselves (the backend devs). For our end users, we take into consideration things like latency (maybe having a single mutation that did two things would get results to the user faster than two mutations that had to be executed serially) or the clarity of error types. For ourselves, we take into consideration how difficult our schema will be to run, secure, and update. For instance, we might decide not to include a query field that would take too much server resources to resolve. Or we might structure parts of the schema to make it easier to add fields later on.

Once we've read this section, we can have a meeting with our frontend devs, UX designers, product managers, etc., to create:

- The core types and queries, based on what data the frontend needs.
- Mutations, based on the user action flows.

We do *not* want to start writing the schema based on backend implementation / naming / structure / tech details. It shouldn't look like our REST APIs or mirror our database tables.

One good option for how to structure your schema creation meeting is event storming, a process from [domain-driven design](#) described in [this article](#).

Our schema also shouldn't be perfect or comprehensive. It should only cover the use cases for which it's needed right now—we shouldn't design it based on hypothetical future requirements:

Fields shouldn't be added to the schema speculatively. Ideally, each field should be added only in response to a concrete need by a consumer for additional functionality, while being designed for maximum reuse by other consumers that have similar needs.

Updating the graph should be a continuous process. Rather than releasing a new “version” of the graph periodically, such as every 6 or 12 months, it should be possible to change the graph many times a day if necessary. New fields can be added at any time. To remove a field, it is first deprecated, and then removed when no consumers use it. —[Principled GraphQL](#)

## Easy to understand

We want others to be able to understand our schema just by reading it. We don't want them to read it, not fully get it, and then have to talk to us or learn through trial and error. Ideally we don't even want them to have to read schema descriptions—just the types themselves. It's the same reason why it's easier to understand readable code than commented code. For example:

```
const resolvers = {
  Mutation: {
    addWineToCart(_, { wineId }, { user }) {
      // first check if user is allowed to drink
      if (new Date(Date.now() - user.dateOfBirth.getTime()).getUTCFullYear() - 1970 < 21) {
        throw new ForbiddenError()
      }
      ...
    }
  }
}
```

```

    }
  }
}

```

The `if` statement condition is complicated and not *readable* (i.e., we don't immediately understand what it means by glancing at it), so we read the comment above it to learn what the `if` statement does. In the below code, however, we can just read it:

```

const US_DRINKING_AGE = 21

const context = async ({ req }) => {
  const user = await getUser(req.headers.authorization)

  user.age = function() {
    const millisecondsSinceBirth = Date.now() - this.dateOfBirth.getTime()
    return new Date(millisecondsSinceBirth).getUTCFullYear() - 1970
  }
  user.isAllowedToDrink = function() {
    return user.age() >= US_DRINKING_AGE
  }

  return { user }
}

const resolvers = {
  Mutation: {
    addWineToCart(_, { wineId }, { user }) {
      if (!user.isAllowedToDrink()) {
        throw new ForbiddenError()
      }
    },
    ...
  }
}

```

While this is many more lines of code, that's not as important as readability. And all we need to read now is `if (!user.isAllowedToDrink())`, which is readily understandable. At most, we may need to mentally move the location of the “not” from “if not user is allowed to drink” to “if user is not allowed to drink.”

For a schema example of this concept, let's imagine we were building an online store, and we had this mutation:

```

type Mutation {
  add(productId: ID!): Cart
  checkout: Order
}

```

Then we realized that while people could probably infer the `add` mutation meant add a product to the cart (given the argument name and return type), it would be clearer if we added a field description:

```

type Mutation {
  # add product to cart
  add(productId: ID!): Cart
  checkout: Order
}

```

While the new “add product to cart” description now appears in Playground autocomplete (and in the DOCS tab after clicking `add`), it has a couple downsides:

- It takes us another step to look for and read the description, versus just reading the field name.
- When we read a query document in the client code, we only see the mutation name—not the description.

We can remove the need for a comment by making the mutation name clearer:

```

type Mutation {
  addProductToCart(productId: ID!): Cart
  checkout: Order
}

```

Readability starts with giving clear names to things. In this case, it was giving a full, specific name—not just `add` or `addProduct`, but `addProductToCart`. Here are a few more examples of specificity:

- Instead of just a `Review` type, use `ProductReview`. Then schema readers know what the review is for, and in the future, we can add other review types, like `StoreReview`, without causing confusion.
- If we have two types of reviews, we shouldn't try to fit them both into a single type. Instead of `Review`, with the 3rd field for product reviews and the 4th and 5th fields for store reviews, we should have two types with different fields:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Review {
  id: ID!
  stars: Int!
  productReviewText: String
  storeDeliveryRating: Int
  storeCustomerSupportRating: Int
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type ProductReview {
  id: ID!
  stars: Int!
  text: String!
}

type StoreReview {
  id: ID!
  stars: Int!
  deliveryRating: Int!
  customerSupportRating: Int
}

```

And if we want to handle them together, we could have them both implement a `Review` interface and reference it:

```

type Query {
  searchReviews(term: String!): [Review!]!
}

interface Review {
  id: ID!
  stars: Int!
}

type ProductReview implements Review {
  id: ID!
  stars: Int!
  text: String!
}

type StoreReview implements Review {
  id: ID!
  stars: Int!
  deliveryRating: Int!
  customerSupportRating: Int
}

```

- Instead of a generic query with a generic argument or a list of optional arguments, make multiple specific queries with non-null arguments:

```
# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Query {
  user(fields: UserFieldInput): User!
}

input UserFieldInput {
  id: ID
  username: String
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Query {
  userById(id: ID!): User!
  userByUsername(username: String!): User!
}
```

- The Guide schema uses a `Date` type for milliseconds since epoch. However, it would be more specific to call it a `DateTime`, since it includes both the date and the time. That would allow us to add `Date` (e.g., `1/1/2000`) and `Time` (e.g., `13:37`) types in the future. It would also be clearer for devs who are used to systems that handle both Dates and DateTimes.

Using specific naming is part of a broader category of being explicit—we want to know what fields and types mean, how to use them, and how they behave, without guessing or trial and error. Here are a few further areas in which we can be explicit:

- Using custom scalars instead of default scalars. Instead of `createdAt: Int`, `createdAt: DateTime`. Instead of `phone: String`, `phone: PhoneNumber`. It explicitly shows what type of value it is, and we can trust that the [custom scalar code](#) will validate `DateTime`s and `PhoneNumber`s wherever they're used in the schema.
- Include default arguments:

```
type Query {
  reviews(
    skip: Int = 0,
    limit: Int = 10,
    orderBy: ReviewOrderBy = createdAt_DESC
  ): [Review!]!
}

enum ReviewOrderBy {
  createdAt_ASC
  createdAt_DESC
}
```

- Use non-null (`!`) to explicitly denote which values will always be returned, or which arguments are required. However, in some cases it's better to not use it:
  - If clients use multiple root query fields in a single document, then leave them all nullable, because if one is non-null and null is returned (e.g., due to an error), it will [null cascade](#) all the way up to a `{ "data": null }` response, which will prevent the client from receiving the other root query fields.
  - If there's any chance a field will occasionally not be available, for instance a `User.githubRepositories` field whose resolver relies on the GitHub API being accessible, make it null. We do this so that when we can't reach the GitHub API (their servers are down, or there's a network issue, or we hit our API quota, for example), queries for user data can receive the other fields.
- Build expected errors into the schema. Then devs will know what error responses look like and will be able to handle them more easily than if they were in the `"errors"` JSON response property.
  - In the below [Mutations](#) section, we'll include expected errors in the response type.
  - Earlier in the [Union errors](#) section, we included deleted and suspended users in the search results:

```
type Query {
```

```

    searchUsers(term: String!): [UserResult!]!
  }

  union UserResult = User | DeletedUser | SuspendedUser

```

- We can also prevent errors from happening with our schema structure. For instance, if there are some queries that are public and some for which the client must be logged in, we can prevent them from them receiving unauthorized errors by having the public queries as root fields and the logged-in queries as `viewer` fields:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Query {
  me: User
  teams: [Team]

  # must be logged in
  projects: [Project]

  # must be logged in
  reports: [Report]
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Query {
  me: Viewer
  teams: [Team]
}

type Viewer {
  id: ID
  name: String
  projects: [Project]
  reports: [Report]
}

```

Only when we can't make a meaning or behavior explicit should we add a description to the schema.

Lastly, a couple more things that are helpful for readability:

- Consistency in naming. For instance, how we name queries for a single item versus a list:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Query {
  project(id: ID): Project
  projects: [Project]

  getReport(id: ID): Report
  listReports: [Report]
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Query {
  project(id: ID): Project
  projects: [Project]

  report(id: ID): Report
  reports: [Report]
}

```

Or the verbs we use with mutations:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Mutation {

```

```

    deleteProject(id: ID!): DeleteProjectPayload
    removeReport(id: ID!): RemoveReportPayload
  }

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Mutation {
  deleteProject(id: ID!): DeleteProjectPayload
  deleteReport(id: ID!): DeleteReportPayload
}

```

- Grouping fields into sub-objects: When a group of fields are related, we can create a new object type. Imagine our reviews had comments that rated the helpfulness of the review:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Review {
  id: ID!
  text: String!
  stars: Int
  commentCount: Int!
  averageCommentRating: Int
  averageCommentLength: Int
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Review {
  id: ID!
  text: String!
  stars: Int
  commentStats: CommentStats!
}

type CommentStats {
  count: Int!
  averageRating: Int
  averageLength: Int
}

```

## Easy to use

While ease of use is determined largely by ease of understanding, there are other factors that can contribute:

- Include fields that save the client from having to go through computation, logic, or other processing. For instance, we provide `Review.fullReview`:

```

const resolvers = {
  Review: {
    fullReview: async (review, _, { dataSources }) => {
      const author = await dataSources.users.findOneById(
        review.authorId,
        USER_TTL
      )
      return `${author.firstName} ${author.lastName} gave ${review.stars} stars, saying: "${review.text}"`
    },
  },
}

```

If the client wants the whole review text in a sentence like that, they could construct it themselves by querying for all the pieces of information and putting it together. Instead, we do it for them, saving them the effort. Similarly, if our clients often want the total comment count, we can include that in the connection so they don't have to do the work of requesting all the comments and counting them:

```

type Review {

```



```

    id: ID!
    text: String!
    comments: CommentsConnection!
  }

  type CommentsConnection {
    nodes: [Comment]
    totalCount: Int!
  }

```

Or, if we have a purchasing app where orders have complex states and business logic, we could include a `readyForSubmission` field so the client doesn't have to write the logic code:

```

type Order {
  id: ID!
  ...
  readyForSubmission: Boolean!
}

```

- Make fields easy to use. For instance when dealing with money, fractional amounts are often more difficult to work with than integers, so we can provide `Int` fields:

```

# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
type Charge {
  dollars: Float!
}

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thumbsup.png' title=':thumbsup:' />
type Charge {
  cents: Int!
}

```

- If we have a public API for third parties, then we can make their integration easier by supporting their preferred libraries. In the case of GraphQL, the only common library with schema requirements is Relay. The [list of requirements](#) includes the cursor connections we [discussed earlier](#), a particular structure to mutations, and a common `Node` interface for object types:

```

interface Node {
  id: ID!
}

type User implements Node {
  id: ID!
  firstName: String!
}

type Review implements Node {
  id: ID!
  text: String!
}

```

## Mutations

As with the rest of the schema, the first thing to think about for mutations is their names. While some choose to do `typeVerb` (like `reviewCreate`, `reviewUpdate`, and `reviewDelete`) so that GraphQL's alphabetical schema docs will group mutations by type, we recommend the more readable `verbType`: `createReview`, `updateReview`, and `deleteReview`. And, as mentioned before, we recommend verb consistency—so for example, using `deleteUser` instead of `removeUser` to match `deleteReview`.

However, we don't recommend uniformly implementing `create|update|delete` mutations for each type. Instead, provide mutations according to the needs of the client—which actions will they be performing? In some cases, types are never deleted, or they're created automatically, or the update step should be named something else or should happen in stages. For instance, imagine a store checkout process in which the server needs to do something (save data, validate, talk to an API, etc.) for each of these steps:

- Create a cart.
- Add products to the cart.
- Apply a coupon code.
- Add shipping address.
- Add payment information.
- Submit order.

We could have the client use `createCart` for the first step and a single generic `updateCart` mutation for each of the rest. (First they'd call `updateCart(productId)`, and then `updateCart(couponCode)`, etc.) However, it would require a large amount of optional arguments, and we would have to write a long field description telling the dev which arguments to use in which order. Instead, we should write multiple mutations with specific names:

```
type Mutation {
  createCart: Cart!
  addProductsToCart(input: AddProductsToCartInput): Cart!
  applyCoupon(input: ApplyCouponInput): Cart!
  addShippingAddressToCart(input: AddShippingAddressToCartInput): Cart!
  addPaymentToCart(input: AddPaymentToCartInput): Cart!
  createOrder(cartId: ID!): Order!
}

input AddProductsToCartInput {
  cartId: ID!
  productIds: [ID!]!
}

input ApplyCouponInput {
  cartId: ID!
  code: String!
}

input AddShippingAddressToCartInput {
  cartId: ID!
  address: AddressInput!
}

input AddPaymentToCartInput {
  cartId: ID!
  payment: PaymentMethodInput!
}
```

- For most of the mutations, we end with `ToCart` to be specific. Just `addProducts` could be adding them to a wishlist, or `addPayment` could be adding a payment method to your account. And if there's anything besides a cart to which a coupon might be applied in the future, we should change `applyCoupon` to `applyCouponToCart`!
- We do `addProductsToCart` instead of the singular `addProductToCart` in case the client might want to add multiple products at a time (it's easier to send a single mutation with an array of IDs than a single-ID mutation many times).

## Arguments

The most common pattern for mutation arguments is a single input object type. Some people choose to instead have a two-argument limit, when one argument is an ID, like this:

```
type Mutation {
  applyCoupon(cartId: ID!, coupon: String!): Cart!
```

```
addShippingAddressToCart(cartId: ID!, address: AddressInput!): Cart!
}
```

A couple benefits of a single argument are:

- The mutation is more readable with a single input object than with a long list of scalars and input objects.
- The input object is more evolvable (we can't deprecate an argument, but we can deprecate an input object field).

Here are a few more considerations when it comes to mutation arguments:

- Earlier we recommended creating specific scalar types over using built-in generics, but we may want to avoid that for mutation arguments. If we use our own scalar types, then the client may have to go through two requests to discover all the errors. If there are errors in both the scalar validation (for instance, an invalid phone number) and in the business logic (for instance, the order size is too large), then the client's first request will only receive the validation error. When they send a second request with a fixed phone number, they'll receive the business logic error. We can improve the client's experience by allowing them to receive all errors at once, which we do by using `String` instead of our own `PhoneNumber` scalar, and doing both the phone number validation and the business logic checks in our resolver code. Then our resolver can return all the errors together. We also have more flexibility on how we return the error—a scalar validation error shows up in the `"errors"` attribute of the JSON response, whereas in our resolver, we can either throw an error *or* return an error—an option we'll see in the next section.
- The client can generate and provide a unique `clientMutationId` for mutations they want to make sure are *idempotent*—that don't get executed multiple times. For instance, if the client sent the below mutation and then lost internet connection and resent, the server could receive the mutation a second time once the connection is back. To avoid this issue, our server code could check to see if the `clientMutationId` on the second mutation matches the first. If it does, our code won't process the second mutation.

```
mutation {
  buyStock(input: { ticker: "TSLA", shares: 10, clientMutationId: "mvvAb9sDGnPYNtZm" }) {
    id
  }
}
```

```
type Mutation {
  buyStock(input: BuyStockInput): Order!
}

input BuyStockInput {
  ticker: String!
  shares: Int!
  clientMutationId: ID!
}
```

- While it's tempting to [DRY](#) our code by sharing input types between create and update mutations, we don't recommend it. We have to use at least one non-null field for the ID (since it's not used during creation), and we have to make all fields non-null if we want to be able to provide the update mutation with just the fields we want to change. However, doing that removes the clarity around which fields are required when creating.

```
# <img align='absmiddle' alt=':-1:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/-1.png' title=':-1:' />
mutation {
  createReview(input: ReviewInput!): Review!
  updateReview(input: ReviewInput!): Review!
}

input ReviewInput {
  # only provide when updating
  id: ID!
  # required when creating
}
```

```

    text: String
    stars: Int
  }

# <img align='absmiddle' alt=':thumbsup:' class='emoji' src='/gitbook/gitbook-plugin-advanced-emoji/emojis/thum
bsup.png' title=':thumbsup:' />
mutation {
  createReview(input: CreateReviewInput!): Review!
  updateReview(input: UpdateReviewInput!): Review!
}

input CreateReviewInput {
  text: String!
  stars: Int
}

input UpdateReviewInput! {
  id: ID!
  text: String
  stars: Int
}

```

## Payloads

So far our mutations have been returning the object they alter or throwing errors. For instance, `createReview` might return a `Review` object or throw an `InputError` that's serialized in the response JSON's `"errors"` attribute.

However, there are a couple issues with this:

- Returning a single type is inflexible—what if multiple types are altered during the mutation, or we want to provide the client with more information about how the mutation went?
- As we discussed in [Union errors](#), it's better to return expected errors than to throw them: It's easier for client code to handle, and it documents the possible errors and their associated data (whereas thrown errors like the `InputError` we created are undocumented / do not appear in the schema).

We solve both of these issues by returning a payload type:

```

type Mutation {
  createReview(input: CreateReviewInput): CreateReviewPayload
}

type CreateReviewPayload {
  review: Review
  user: User
  errors: [Error!]!
}

type Error {
  message: String!
  code: ErrorCode
  field: Field
}

```

When we create a review, our `User.reviews` changes. We can include the user in the payload so that the client can easily update their cached user object. We make both the `review` and `user` optional because we might instead return `errors`. The client's operation would look like:

```

mutation {
  createReview(input: { text: "", stars: 6 }) {
    review {
      id
      text
      stars
      createdAt
    }
    user {

```

```

    reviews {
      id
    }
  }
  errors {
    message
    code
    field
  }
}
}

```

And the response would be:

```

{
  "data": {
    "createReview": {
      "errors": [{
        "message": "Text cannot be empty",
        "code": 105,
        "field": "input.text"
      }, {
        "message": "Stars must be an integer between 0 and 5, inclusive",
        "code": 106,
        "field": "input.stars"
      }]
    }
  }
}

```

In cases when the mutation alters an unknown set of types, we can use the Query type to allow the client to get back whatever data they'd like after the mutation is complete:

```

type Mutation {
  performArbitraryOperation(operation: ArbitraryOperation): PerformArbitraryOperationPayload
}

type CreateReviewPayload {
  query: Query
  errors: [Error!]!
}

```

## Versioning

Most APIs change over time. We can deploy *backward-compatible* changes at any time. We usually try to avoid making *breaking* changes, i.e., changes that may break client code using that part of the API. However, sometimes we want to make a breaking change because it would be a significant improvement. If our API is only used by our clients, and all our clients are web apps, then we can publish a new version of the client at the same time as a breaking API change, and we can force all the currently loaded webpages (now out of date) to reload, and nothing will be broken. However, if we don't want to force-reload our web app, or if we have mobile apps (which we can't force-reload), or if we have a public API (which is used by third parties, whose code we don't have control over), then we have two options:

- **Global versioning.** Publish a new version of the API at a different URL, like `api.graphql.guide/v2/`. Then clients using the original URL will continue to work.
- **Deprecation:**
  - Add a deprecation notice so that, going forward, devs don't use the field.
  - Notify existing API consumers of the deprecation so they can change their code.
  - Monitor the usage of the field.
  - When the field usage falls under a tolerable threshold (number of will-be-broken requests), remove it.

Here are a couple examples of deprecation:

```
type User {
  id: ID!
  name: String @deprecated(
    reason: "Replaced by field `fullName`"
  )
  fullName: String
}

type Mutation {
  createReview(text: String!, stars: Int): Review @deprecated(
    reason: "Replaced by field `createReviewV2`"
  )
  createReviewV2(input: CreateReviewInput): CreateReviewPayload
}
```

While only the deprecation option includes making the breaking change as a step, it usually eventually happens for global versioning as well. There is always a cost of maintaining the old code—whether the code is backing an earlier global version or a deprecated field—and at some point, that cost outweighs the cost of breaking old clients. For instance, we could have a globally versioned API that’s currently on version 5, and almost all of the clients are using v2–v5, and we decide that we’d rather break the few clients still using v1 than continue maintaining it.

We recommend using the deprecation process (also called **continuous evolution**) in lieu of versioning. The downside of deprecating is the schema can get cluttered with deprecated fields. The downside of versioning is the large cost of maintaining old server versions and the increased time it takes to make changes. Given the complexity of deploying and maintaining a new version of the API, we batch changes and create new versions infrequently, whereas we can deprecate at any time.

There are a few reasons why continuous evolution is the better practice compared to versioning, which was common with REST APIs:

- Adding is backward compatible. With REST APIs that don’t have control over what data is returned from an endpoint, any changes, even returning more data than the client expects, can be breaking. With GraphQL APIs, adding a new field doesn’t affect current clients—they only receive the fields specified in their query document.
- Deprecation is built into the GraphQL spec, and GraphQL tooling will show developers when they’re using a deprecated field, so clients will update their code more easily and sooner.
- Since all the fields requested are in the query document, we can know how many clients are using deprecated fields. If we added a `fullName` field to the user REST endpoint, we wouldn’t know how many clients were still using the `name` field. With GraphQL, we know!

We can currently deprecate fields and enum values, and deprecating arguments and input fields will likely be added to the spec in the near future.

We deprecate a field instead of removing it because removing a field is a breaking change. But there are other breaking changes to watch out for as well:

- Removing fields, enum values, union members, or interfaces.
- Changing the type of a field.
- Making an argument or input field non-null.
- Adding a new non-null argument or input field.
- Making a non-null argument nullable.
- Changing a field from non-null to nullable isn’t automatically breaking, but if the server ever does return null for that field, the client can break.

Finally, it’s possible to break clients by adding new enum values, union members, and interface implementations if the client logic depends on all the data they receive fitting their (outdated) set of values/members/implementations.

Ideally, clients will always leave open the possibility that those things could be added.

## Custom schema directives

Background: [Directives](#)

If you're jumping in here, `git checkout 25_0.1.0` (tag [25\\_0.1.0](#), or compare [25...directives](#))

Apollo Server includes the [default directives](#) `@deprecated`, `@skip`, and `@include`. `@skip` and `@include` are *query directives*, so they don't appear in our schema; instead, they're included in query documents and can be used on any field. `@deprecated` is a *schema directive*, and when we add it after a field or enum value in our schema, the directive will be included in responses to introspection queries.

We can make our own schema directives in Apollo Server. When we add them to specific places in our schema, those parts of the schema are modified or evaluated differently when resolving requests. Three examples we'll code are `@tshirt`, which modifies an enum value's description; `@upper`, which takes the result of a field resolver and returns the uppercase version instead; and `@auth`, which throws an error if the user isn't authorized to view that object or field.

- [@tshirt](#)
- [@upper](#)
- [@auth](#)

### @tshirt

Schema directives are implemented by subclassing `SchemaDirectiveVisitor` and overriding one or more methods of the format `visitFoo()`, where `Foo` is the part of the schema to which the directive is applied. Possible parts of the schema are:

- Whole schema
- Scalar
- Object
- Field definition
- Argument definition
- Interface
- Union
- Enum
- Enum value
- Input object
- Input field definition

For example, if it were applied to an enum value:

`src/schema/schema.graphql`

```
directive @tshirt on ENUM_VALUE

enum Package {
  BASIC
  PRO
  FULL @tshirt
  TRAINING @tshirt

  # Group license.
  TEAM @tshirt
}
```

Then our subclass would override `visitEnumValue()`:

`src/directives/TshirtDirective.js`

```
import { SchemaDirectiveVisitor } from 'apollo-server'

class TshirtDirective extends SchemaDirectiveVisitor {
  visitEnumValue(value) {
    ...
    return value
  }
}
```

To determine the structure of `value`, we can either use `console.log()` or look up the type definition of an enum value in the `graphql-js` library. All type definitions are in `src/type/definition.js`, where we can find:

```
export type GraphQLEnumValue /* <T> */ = {
  name: string,
  description: ?string,
  value: any /* T */,
  isDeprecated: boolean,
  deprecationReason: ?string,
  extensions: ?ReadOnlyObjMap<mixed>,
  astNode: ?EnumValueDefinitionNode,
};
```

`isDeprecated` and `deprecationReason` are the fields that are used by the `@deprecated` directive.

It has an optional `description` field, to which we can add a note about T-shirts 🧐:

`src/directives/TshirtDirective.js`

```
import { SchemaDirectiveVisitor } from 'apollo-server'

export default class TshirtDirective extends SchemaDirectiveVisitor {
  visitEnumValue(value) {
    value.description += ' Includes a T-shirt.'
    return value
  }
}
```

Then we need to get it to `ApolloServer()`:

`src/directives/index.js`

```
import TshirtDirective from './TshirtDirective'

export default {
  tshirt: TshirtDirective
}
```

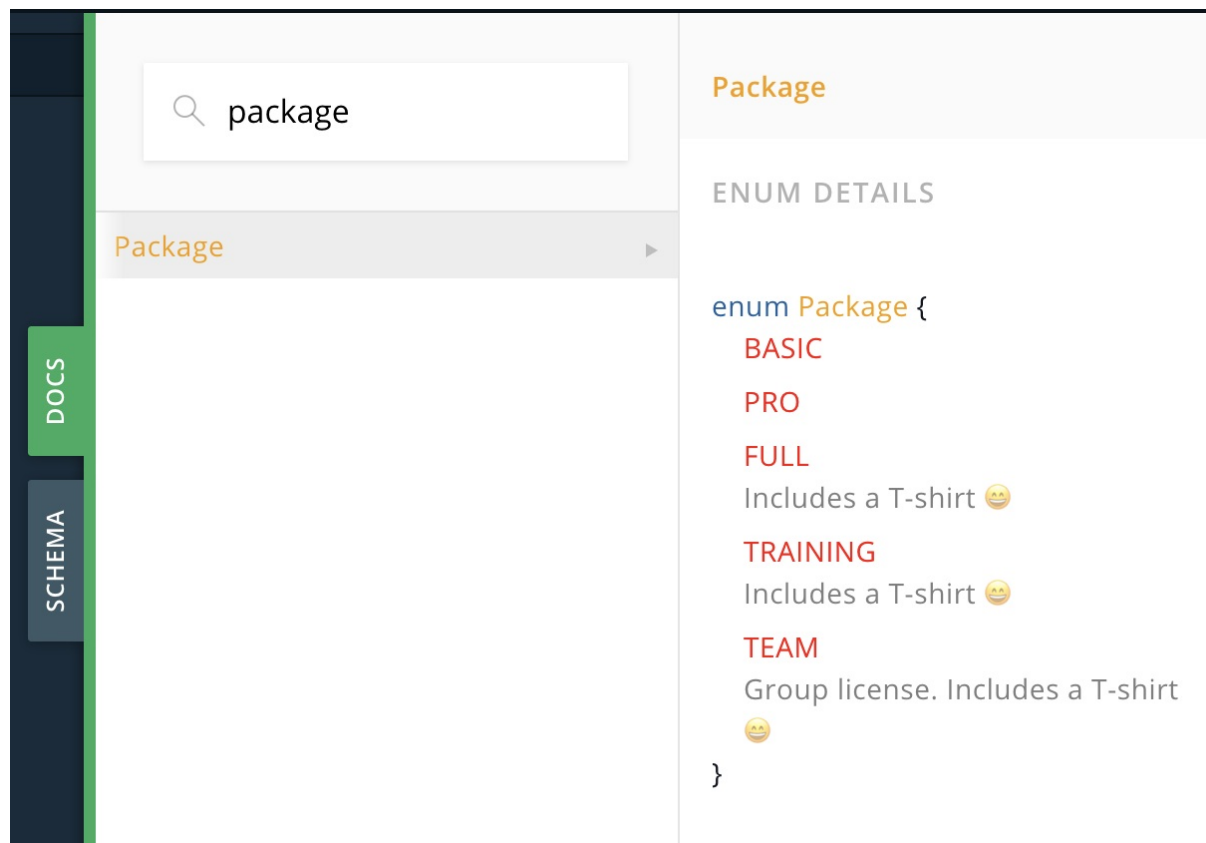
`src/index.js`

```
import schemaDirectives from './directives'

const server = new ApolloServer({
  typeDefs,
  schemaDirectives,
  resolvers,
  dataSources,
  context,
  formatError
})
```

Now we can check the description by using the search box inside Playground's docs tab:





## @upper

When we're making a directive to use on fields, oftentimes what we want to do is call the resolver and modify the result, like this:

```
import { SchemaDirectiveVisitor } from 'apollo-server'
import { defaultFieldResolver } from 'graphql'

class MyDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field) {
    const { resolve = defaultFieldResolver } = field
    field.resolve = async function(...args) {
      const result = await resolve.apply(null, args)
      // modify result
      // ...
      return result
    }
  }
}
```

Here we override the `visitFieldDefinition()` function, which receives a `field` object that has a `resolve` property:

```
export type GraphQLField<
  TSource,
  TContext,
  TArgs = { [argument: string]: any, ... },
> = {
  name: string,
  description: ?string,
  type: GraphQLOutputType,
  args: Array<GraphQLArgument>,
  resolve?: GraphQLFieldResolver<TSource, TContext, TArgs>,
  subscribe?: GraphQLFieldResolver<TSource, TContext, TArgs>,
  isDeprecated: boolean,
  deprecationReason: ?string,
```

```

    extensions: ?ReadOnlyObjMap<mixed>,
    astNode: ?FieldDefinitionNode,
  }
};

```

We redefine `field.resolve`, calling the original `resolve` or the `defaultFieldResolver`, which resolves the field as a property on the parent object when there is no resolver function (e.g., `User: { firstName: (user, _, context) => user.firstName }`). Then we modify and return the result.

Let's use this format to implement an `@upper` resolver, which transforms the result to uppercase:

`src/schema/schema.graphql`

```

directive @upper on FIELD_DEFINITION

type Query {
  hello(date: Date!): String! @upper
  isoString(date: Date!): String!
}

```

And now, since we can't convert an emoji to uppercase, we need `Query.hello` to return lowercase ASCII:

`src/resolvers/index.js`

```

const resolvers = {
  Query: {
    hello: () => 'world ',
    ...
  }
}

```

As above, we redefine the field's `resolve` function, calling the original. This time we check if the result is a string and call `.toUpperCase()`:

`src/directives/UppercaseDirective.js`

```

import { SchemaDirectiveVisitor } from 'apollo-server'
import { defaultFieldResolver } from 'graphql'

export default class UppercaseDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field) {
    const { resolve = defaultFieldResolver } = field
    field.resolve = async function(...args) {
      const result = await resolve.apply(this, args)
      if (typeof result === 'string') {
        return result.toUpperCase()
      }
      return result
    }
  }
}

```

We include the directive class by adding it to this object, where the key corresponds with the directive name `@upper`:

`src/directives/index.js`

```

import TshirtDirective from './TshirtDirective'
import UppercaseDirective from './UppercaseDirective'

export default {
  tshirt: TshirtDirective,
  upper: UppercaseDirective
}

```



## @auth

Directives can also take arguments, which can be scalars, enums, or input object types. `@deprecated`, for instance, takes a `reason` argument of type `String`:

```
type User {
  firstName
  first_name: String @deprecated(reason: "Use `firstName`.")
}
```

We'll be implementing a directive that takes an enum argument:

`src/schema/schema.graphql`

```
directive @auth(
  requires: Role = ADMIN,
) on OBJECT | FIELD_DEFINITION

enum Role {
  USER
  MODERATOR
  ADMIN
}
```

Our `@auth` directive is for specifying which objects or fields ( `on OBJECT | FIELD_DEFINITION` ) require a `Role`. If the `requires` argument isn't used, then the default `ADMIN` is used.

Our `AuthDirective` class is similar to `UppercaseDirective` in that we're wrapping the `field.resolve()` function in a new function. However, instead of modifying the result, our wrapping function throws an error if the current user's role doesn't match the required role:

`src/directives/AuthDirective.js`

```
import { SchemaDirectiveVisitor, ForbiddenError } from 'apollo-server'
import { defaultFieldResolver } from 'graphql'

export default class AuthDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field) {
    const { resolve = defaultFieldResolver } = field
    field.resolve = (...resolverArgs) => {
      const requiredRole = this.args.requires
      const context = resolverArgs[2]

      if (!context.user.roles.includes(requiredRole)) {
        throw new ForbiddenError(`You don't have permission to view this data.`)
      }

      return resolve.apply(null, resolverArgs)
    }
  }
}
```

The directive's arguments are available at `this.args.*. resolverArgs[2]`, the third argument passed to resolvers, is always the context where we put the user doc. We assume that the user's roles are stored in the user doc as an array of strings (like `roles: ['USER']` or `roles: ['USER', 'ADMIN']`).

Since `@auth` works on `OBJECT | FIELD_DEFINITION`, we also need to implement the `visitObject()` method. It needs to go through each field in the object and wrap the `resolve()` function. We also need to mark if a field has been wrapped, so that we don't double-wrap (if we use `@auth` on both the object and field `foo` in the object, `visitObject()` will wrap all fields, and then `visitFieldDefinition()` will wrap `foo`, which has already been wrapped).

```
import { SchemaDirectiveVisitor } from 'apollo-server'
import { defaultFieldResolver } from 'graphql'

export default class AuthDirective extends SchemaDirectiveVisitor {
  visitObject(objectType) {
    objectType._requiredRole = this.args.requires

    const fields = objectType.getFields()
    Object.keys(fields).forEach(fieldName => {
      const field = fields[fieldName]
      this._wrapResolveFn(field, objectType)
    })

    objectType._wrappedResolveFn = true
  }

  visitFieldDefinition(field, { objectType }) {
    field._requiredRole = this.args.requires

    const alreadyWrapped = objectType._wrappedResolveFn
    if (!alreadyWrapped) {
      this._wrapResolveFn(field, objectType)
    }
  }

  _wrapResolveFn(field, objectType) {
    const { resolve = defaultFieldResolver } = field

    field.resolve = (...args) => {
      const requiredRole = field._requiredRole || objectType._requiredRole
      const context = args[2]

      if (!context.user.roles.includes(requiredRole)) {
        throw new Error('not authorized')
      }

      return resolve.apply(null, args)
    }
  }
}
```

We save the required role on the field and the object so that inside the wrapper, we can determine which to use (preferencing a role saved on the field over one saved on the object):

```
const requiredRole = field._requiredRole || objectType._requiredRole
```

We use underscores for data we save (`._requiredRole` and `._wrappedResolveFn`) and for the method we define (`._wrapResolveFn()`) to indicate they're private (not meant to be used / called by code outside this class).

Note that `visitFieldDefinition()` receives a second argument with that field's object type. Here are [all the methods](#) that have second arguments:

- `visitFieldDefinition(field, { objectType })`
- `visitArgumentDefinition(argument, { field, objectType })`

- `visitEnumValue(value, { enumType })`
- `visitInputFieldDefinition(field, { objectType })`
- `visitSchema(schema, visitorSelector)` (see [explanation of visitorSelector](#))

Finally, let's add our new directive class to our server:

src/directives/index.js

```
import TshirtDirective from './TshirtDirective'
import UppercaseDirective from './UppercaseDirective'
import AuthDirective from './AuthDirective'

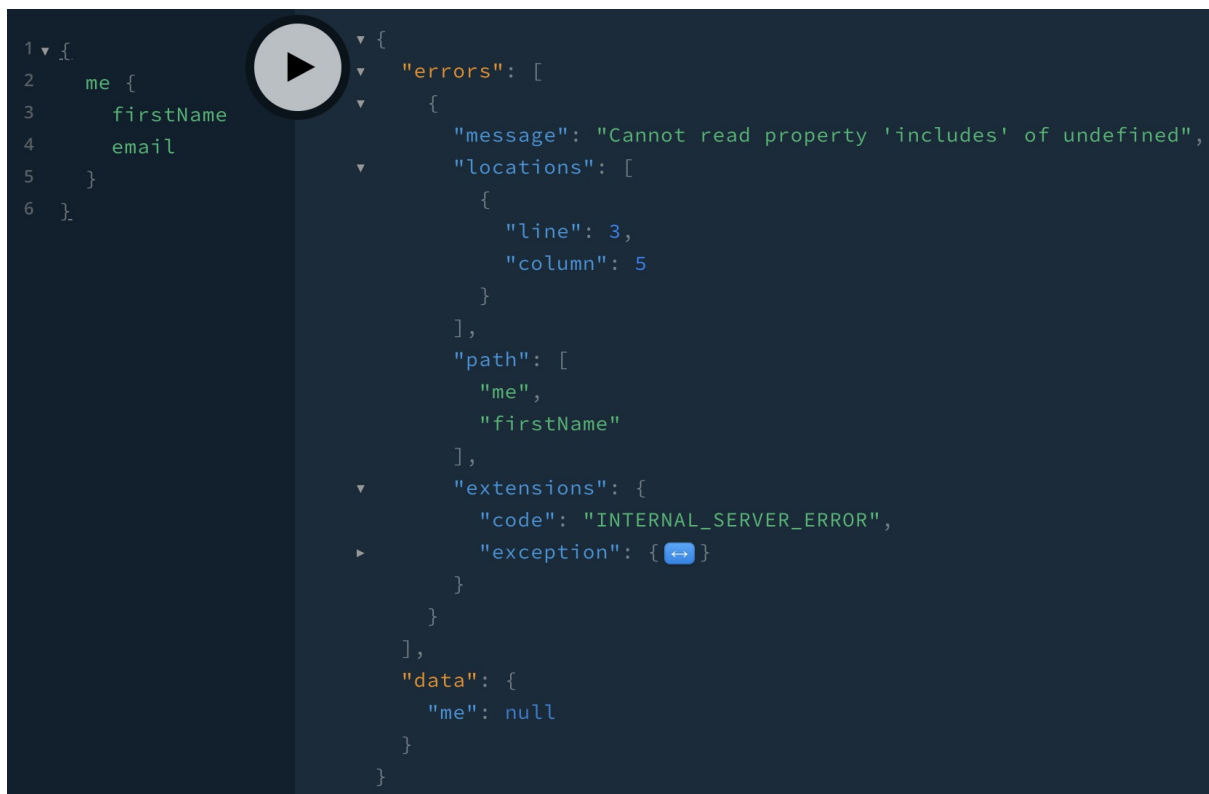
export default {
  tshirt: TshirtDirective,
  upper: UppercaseDirective,
  auth: AuthDirective
}
```

Now we can test out the directive:

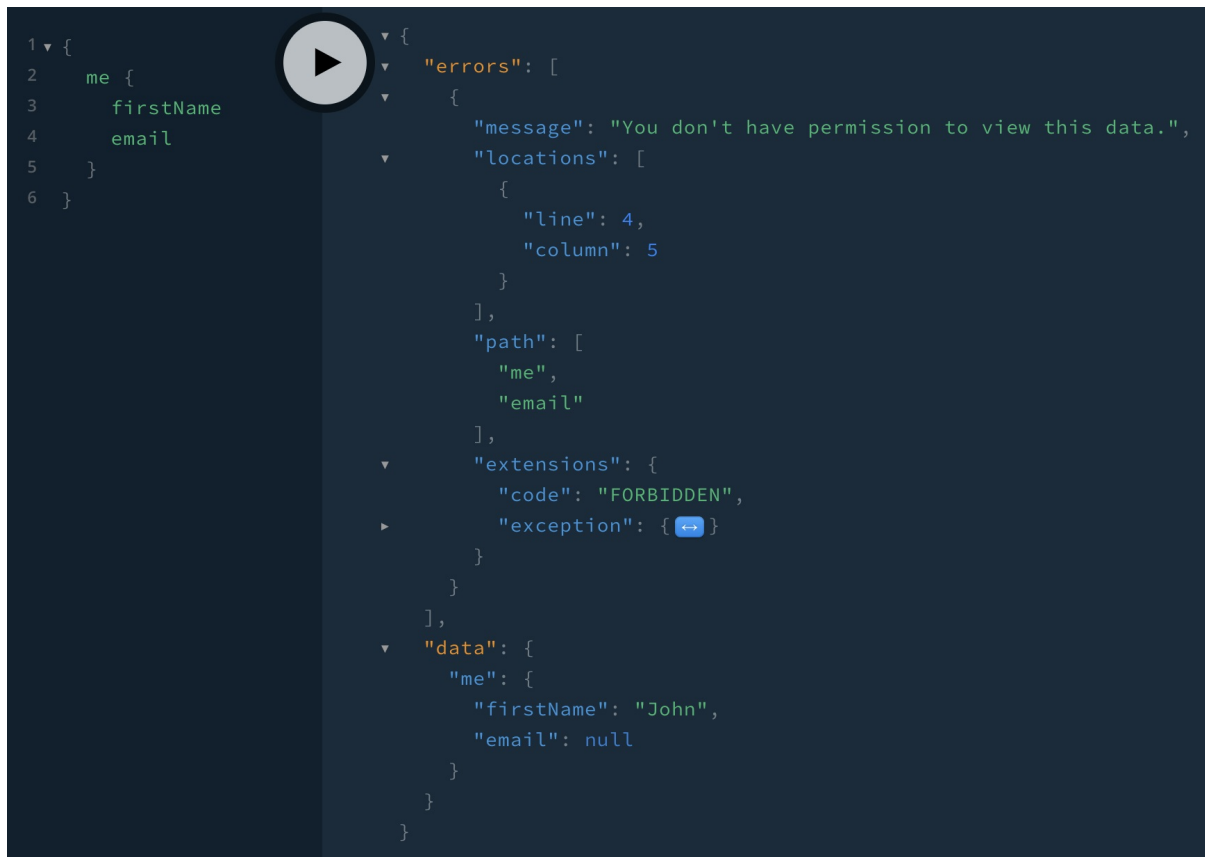
src/schema/User.graphql

```
type User @auth(requires: USER) {
  id: ID!
  firstName: String!
  lastName: String!
  username: String!
  email: String @auth(requires: ADMIN)
  photo: String!
  createdAt: Date!
  updatedAt: Date!
}
```

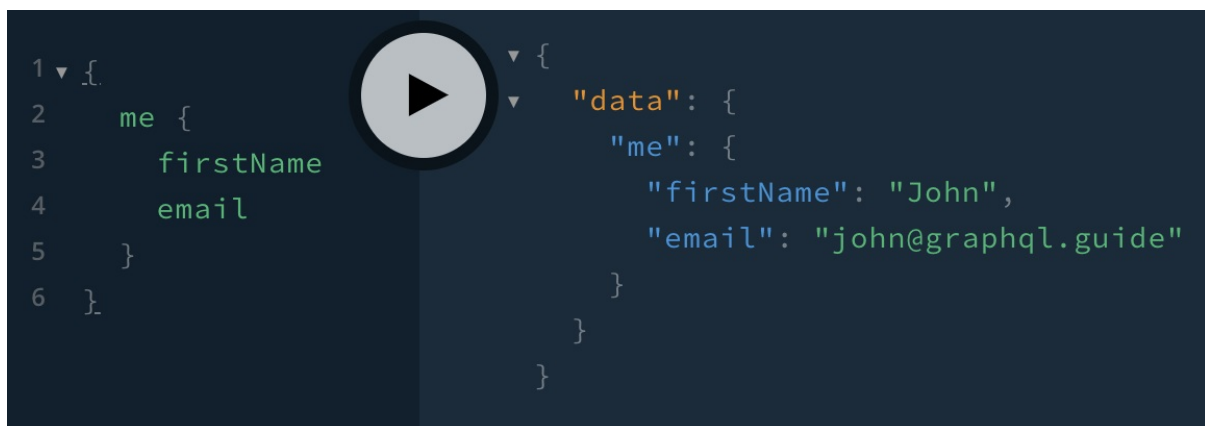
Without a `roles` field on our user doc, we get an error and null data:



With `"roles": ["USER"]` , we get data and an error:



With `"roles": ["USER", "ADMIN"]`, we get all the data:



## Subscriptions in depth

### Server architecture

Back in the [Deployment options](#) section, we decided to deploy to a PaaS because our app has subscriptions, which don't work on FaaS. However, we can split our code into two servers: One that handles subscriptions and WebSockets and runs on a PaaS long-running process, and one that handles queries and mutations over HTTP and runs on a FaaS. This way, our two tasks, which have very different hosting requirements, can be maintained and scaled independently according to their needs.

Let's recall what our subscription code looks like. When the client sends this operation:

```
subscription {
```

```
githubStars
}
```

Our `Subscription.githubStars.subscribe` function is called:

`src/resolvers/Github.js`

```
import { pubsub } from '../util/pubsub'

export default {
  Subscription: {
    githubStars: {
      subscribe: () => pubsub.asyncIterator('githubStars')
    }
  }
}
```

The server now keeps the WebSocket open and sends over it anything that's published to the `githubStars` iterator (`pubsub.publish('githubStars', foo)`).

When our server starts up, we start polling:

`src/index.js`

```
const start = () => {
  Github.startPolling()
  ...
}
```

`src/data-sources/Github.js`

```
export default {
  async fetchStarCount() {
    const data = await githubAPI.request(GUIDE_STARS_QUERY).catch(console.log)
    return data && data.repository.stargazers.totalCount
  },

  startPolling() {
    let lastStarCount

    setInterval(async () => {
      const starCount = await this.fetchStarCount()
      const countChanged = starCount && starCount !== lastStarCount

      if (countChanged) {
        pubsub.publish('githubStars', { githubStars: starCount })
        lastStarCount = starCount
      }
    }, 1000)
  }
}
```

When the number of stars changes, the new count is published to the `githubStars` iterator, and the server sends it out to all the clients who have subscribed.

All the above code can be separated into a new Node server. In fact, since we switched from the default in-memory pubsub to [Redis PubSub](#), the code that publishes updates doesn't need to be in the same process that receives subscriptions and handles WebSockets! So if we wanted, we could have three servers:

- Subscription server: A PaaS that supports WebSockets
- Query and mutation server: FaaS
- `githubStars` publishing server: FaaS with scheduled periodic executions

Usually, most of an app's publishing comes from the mutation server: When a mutation changes data, it publishes the change with the new data. When we're publishing data from an external source, then we need a function triggered on a schedule to check for changes or the source has to notify us when things change (a [webhook](#)). When data is changed from places outside our mutation server, we can publish to our subscriptions in three different ways:

- Have those other places (for instance, a legacy application that works with the same business data) publish the changes they make to Redis.
- Have a long-running server poll the database for changes. This can take a significant amount of memory, since the process needs to keep the current state of the data in order to see what has changed. On the other hand, it scales well with high write loads (since changing data doesn't trigger anything). This is the strategy [Hasura](#) uses.
- Use a special database:
  - [RethinkDB](#) provides *change feeds* as a way to be notified when the results of a query change (though not all possible queries are supported).
  - MongoDB provides an *oplog*—a log of all database operations—that we can have a server listen to (*tail*). If data changes frequently, it can take a significant amount of CPU to process the oplog, determining which operations are changes that should be published for our subscriptions.

In the [Meteor](#) framework, you can use a mix of oplog tailing and polling when oplog tailing is too CPU-intensive.

## Subscription design

Our `githubStars` subscription is basic—just a single scalar value.

```
type Subscription {
  githubStars: Int
}
```

Usually subscriptions are for getting updates to an object or list of objects. For instance, our `createReview` subscription updates clients on objects being added to the list of reviews.

```
type Subscription {
  reviewCreated: Review!
}
```

If we wanted to get all types of updates, we have three options:

1) Adding `reviewUpdated` and `reviewDeleted`:

```
type Subscription {
  reviewCreated: Review!
  reviewUpdated: Review!
  reviewDeleted: ID!
}
```

2) A single `reviews` subscription:

```
type Subscription {
  reviews: ReviewsPayload
}

union ReviewsPayload =
  CreateReviewPayload |
  UpdateReviewPayload |
  DeleteReviewPayload

type CreateReviewPayload {
  review: Review!
}
```



```

type UpdateReviewPayload {
  review: Review!
}

type DeleteReviewPayload {
  reviewId: ID!
}

```

Here we could share the same payloads as the `createReview`, `updateReview`, and `deleteReview` mutations.

3) Calling `reviewCreated` and a `review(id)` subscription for each review loaded on the page:

```

type Subscription {
  reviewCreated: Review!
  review(id: ID!): ReviewPayload!
}

union ReviewsPayload =
  UpdateReviewPayload |
  DeleteReviewPayload

```

Options #1 and #2 are similar in that the client gets updates to the entire list of reviews. In #2, they have to make fewer subscriptions. In #1, they have more flexibility if for some reason they only wanted to subscribe to `reviewCreated` and not the others. In #3, the client makes many more subscriptions, but doesn't have to deal with receiving events about reviews they don't care about. In #1 and #2, unless the user has scrolled enough to load the entire list on the page, they're getting events about review objects that aren't on the page or in the cache, and ignoring them. Given that it takes resources to receive WebSocket messages and check to see if the review is in the cache, we may want to go with #3. In our use case, though, editing and deleting reviews happens infrequently, and even if adding reviews happens frequently, those events are usually all relevant, since the default sort order is most recent. So we might go with the simplicity of #2.

If we had a review detail page that just showed a single review, we would use the `review(id)` subscription. If the page also had a list of comments, then we might do:

```

type Subscription {
  reviewCreated: Review!
  review(id: ID!): ReviewPayload!
  commentsForReview(reviewId: ID!): CommentsPayload!
}

union ReviewsPayload =
  UpdateReviewPayload |
  DeleteReviewPayload |

union CommentsPayload =
  CommentCreatedPayload |
  CommentUpdatedPayload |
  CommentDeletedPayload

```

Of course, if we had (or thought we might have in the future) a different kind of comment elsewhere in our app, we would change all the instances of `Comment*` to `ReviewComment*`.

And if the client was on page `/review/123`, we would subscribe to `review(id: "123")` and `commentsForReview(id: "123")`. As before with the list of reviews, if there might be a lot of comments and comment edit/delete activity, and only some of the comments were shown on the page, we might instead subscribe to updates to each individual comment: `comment(id: "<comment id>")`.

The design of our subscriptions depends on which client views we want realtime updates for, the size of the data set, and the frequency of updates. We take into consideration how much work it takes for the client to make the subscriptions, how much work it takes them to filter out unwanted messages, and also avoiding overfetching data on

the messages we do want. For instance, we return just the ID of a deleted object instead of the whole object. And if we had a granular `changeReviewStars` mutation, we could union and resolve to a `ChangeReviewStarsPayload` type. The client could then only select the `stars` field instead of the whole review:

```
fragment ChangeReviewStars on ChangeReviewStarsPayload {
  review {
    id
    stars
  }
}

fragment CreateReview on CreateReviewPayload {
  review {
    id
    text
    stars
    createdAt
  }
}

fragment DeleteReview on DeleteReviewPayload {
  reviewId
}

subscribe {
  reviews {
    ...ChangeReviewStars
    ...CreateReview
    ...DeleteReview
  }
}
```

## Security

Background: [HTTP](#), [Databases](#), [Authentication](#)

- [Auth options](#)
  - [Authentication](#)
  - [Authorization](#)
- [Denial of service](#)

In this section, we'll start out with an overview of general server-side security and then get to a few topics specific to GraphQL.

Computer security is protecting against:

- Unauthorized actions
- Theft or damage of data
- Disruption of service

Here are a few levels of vulnerability relevant to securing servers from the above threats, along with some methods of risk management:

- **People and their devices:** People that have access to our systems, like employees at our company, hosting companies, and service companies like Auth0.
  - Train employees on security, including avoiding the most common malware avenues: visiting websites and opening files.
  - Avoid personal use of work devices.
  - Install [antivirus](#) on work computers.
  - [Vet](#) employee candidates.
  - Access production systems and data from a limited number of devices that are not used for email or web

browsing.

- **Physical access:** The capability to physically get to servers that store or handle our data.
  - Make sure device hard drives are encrypted with complex login passwords, or locked away when not in use.
  - Assess risk level of our service companies (for example [AWS perimeter security](#)).
- **Network:** Users being able to access our server over the internet or view data in transit.
  - Keep our server IP addresses private.
  - Use a DNS provider that hides our server IPs and handles DDoS attacks (like [Cloudflare](#) or AWS's [Shield Standard, CloudFront, & Route 53](#)).
  - Force HTTPS: When a client makes a connection to our server on port 80 (unencrypted), redirect them to port 443, which will ensure all further data sent between us and the client is encrypted.
- **Operating system:** Hackers exploiting a vulnerability in our server OS (usually Linux).
  - Apply security patches or use a PaaS or FaaS, where OS security is taken care of for us.
- **Server platform:** Node.js.
  - Apply security updates to Node.js, or use a PaaS or FaaS, where security updates are done automatically.
- **Application layer:** GraphQL execution and our code. The following sections cover this area of security.

After we implement protections, we can hire a firm to do a [security audit](#) and use [HackerOne](#) to find areas we didn't sufficiently cover.

Any system can be hacked—it's just a matter of the level of resources put into hacking. The two largest sources relevant to companies are eCrime (criminal hacking—often financial or identity theft) and the Chinese government (stealing trade secrets from foreign companies). Most large companies have been hacked at some point to some degree.

After we have been hacked, it's important to be able to:

1. Figure out how it happened.
2. Ensure the attackers no longer have access.
3. Know what data was accessed.
4. Recover deleted data.

For #1 and #3, we can set up access logs for our production servers, databases, and sensitive services, and for #4, we can set up automatic database backups (MongoDB Atlas has options for either snapshots or continuous backups). Step #2 depends on #1—if one of our service accounts was compromised, we can change the password. If one of our API user's accounts was stolen (session token, JWT, or password), then we need to delete their session or re-deploy with code that blocks their JWT (and if we're using password authentication, delete their current password hash and send a password reset email).

One important way to mitigate the damage of a database hack is hiding sensitive database fields—either by storing only hashes, in the case of passwords, or by storing fields encrypted (using an encryption key that's not stored in the database). Then an attacker won't know the user's password (which they'd likely be able to use to log in to the user's accounts on other sites), and they won't be able to read sensitive data unless they also gain access to the encryption key.

Here are a few application-layer security risks that apply to API servers in general—not just GraphQL servers:

- **Parameter manipulation:** When clients alter operation arguments. We protect against this by checking arguments to ensure they're valid, and by not trusting them (for instance, we should use the `userId` from the context instead of from an argument).
- **Outdated libraries:** Our code depends on a lot of libraries, any of which may have security vulnerabilities that affect our app. For Node.js, we can use `npm audit` to check for vulnerabilities in our libraries.
- **Database injection** like [SQL injection](#) and [MongoDB injection](#)
- **XSS:** On the client, preventing XSS involves sanitizing user-provided data before it's added to the DOM, but on the server, we use a [Content-Security-Policy](#) header.
- **Clickjacking:** Use [X-Frame-Options headers](<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/>)

- Race conditions, especially [TOCTOU](#): Imagine multiple of our servers are running the same mutation from the same user at the same time. We may need to use database transactions or other logic to prevent this type of attack.
- Number processing: Bugs that involve working with numbers, including conversion, rounding, and overflows.

## Auth options

*Auth* is an imprecise term—sometimes it's used to mean authentication, sometimes authorization, and sometimes both. In this case, we mean both:

- [Authentication](#)
- [Authorization](#)

## Authentication

Background: [Authentication](#)

The server receives a JWT or session ID in an HTTP header, which it uses to decode or look up the user. If we're putting our GraphQL server in front of existing REST APIs, then we may want to just pass the header along to the REST APIs—they can continue doing the authentication (and authorization), returning null or returning errors that we can format as GraphQL errors.

However, usually we'll handle user decoding in the GraphQL server. In the case of federation, we decoded the user [in the gateway](#) and passed the object in a `user` header to the services. In the case of our monolith, we decoded [in the context function](#) and provided `context.user` to the resolvers.

But how does the client get the JWT or session ID in the first place? In our case, we used an external service: We [opened a popup](#) to an Auth0 site that did both signup and login and provided the client with a JWT. Other options include:

- Hosting our own identity server (for example the free, open-source [Ory server](#)).
- Adding HTTP endpoints to our GraphQL server (for example with the [Passport library](#)).
- Adding mutations to our GraphQL server (for example the [accounts-js](#) library adds `Mutation.register`, `Mutation.authenticate`, etc. to our schema).
- Using our hosting provider's identity service (for example [Netlify Identity](#) if our server is hosted with [Netlify Functions](#), or [Amazon Cognito](#) with AWS Lambda).

Hosting our own separate identity server might be the most common solution.

## Authorization

After we authenticate the client, we either have their decoded token object (in the case of JWTs) or their user object (in the case of sessions). Both the token and the user object should have the user's permissions. Permissions can be stored in different ways—usually a list of roles or scopes, or, at its most simple, as an `admin` boolean field.

Once we have the user's permission info, our server has to determine which data to allow the user to query and which mutations to allow the user to call. There are a number of different places where we can make this determination:

- **REST services:** In the case of putting a GraphQL gateway in front of existing REST services that already do authorization checks, we can continue to let them do the checks.
- **Context:** If we only want logged-in users to be able to use our API, we can throw an `AuthenticationError` in our `context()` function whenever the HTTP header is missing or the decoding/session lookup fails.
- **Model:** We can do the checks in our data-fetching code. This is the best option when we have both a GraphQL and REST API, both of which call the model code. (This way, we don't have to duplicate authorization checks.)
- **Directives:** We can add directives to fields or types in our schema—for instance, `@isAuthenticated` or

`@hasRoles(roles: [ADMIN])`. A library we can use that defines these directives for us is [graphql-auth-directives](#).

- **Resolvers:** In the server we built in this chapter, we did all our authorization checks in our resolver functions. The biggest downside to this approach is repetition as the schema gets larger—for instance, we’d probably wind up with a lot of `if (!user) { throw new ForbiddenError('must be logged in') }`. It’s also harder to get a broader sense of which parts of the schema have which authorization rules. With directives, we can easily scan through the schema, and with middleware, we can look at the below `shield({ ... })` configuration and see everything together.
- **Middleware:** We can use [graphql-middleware](#)—functions that are called before our resolvers are called. In particular, we can configure the [GraphQL Shield](#) middleware library to run authorization functions before our resolvers like this:

```
const isAuthenticated = rule({ cache: 'contextual' })(
  async (parent, args, context, info) => {
    return context.user !== null
  }
)

const isAdmin = rule({ cache: 'contextual' })(
  async (_, __, context) => {
    return context.user.roles.includes('admin')
  }
)

const isMe = rule({ cache: 'strict' })(
  async (parent, _, context) => {
    return parent._id.equals(context.user._id)
  }
)

const permissions = shield({
  Query: {
    me: isAuthenticated,
    secrets: isAdmin
  },
  Mutation: {
    createReview: isAuthenticated
  },
  User: {
    email: chain(isAuthenticated, isMe)
  },
  Secret: isAdmin
})
```

The equivalent **directives** schema would be:

```
type Query {
  user(id: ID!): User
  me: User @isAuthenticated
}

type Mutation {
  createReview(review: CreateReviewInput!): Review @isAuthenticated
}

type Secret @hasRole(roles: [ADMIN]) {
  key: String
}
```

And for `User.email`, we could either do a resolver check or create a new directive.

In each of the last three authorization locations—**directives**, **resolvers**, and **middleware**—we have to be careful about adding rules only to our root query fields. Since our data graph is interconnected, oftentimes there will be other ways to reach a sensitive type through a connection from another field. So it’s usually necessary to add rules to types, as we do with the `Secret` type above. Unfortunately, we can’t do that in resolvers—just directives and middleware.

## Denial of service

Denial of service is a type of attack in which the attacker overloads our servers' capacity to process requests, resulting in legitimate users being unable to use our app. While some attacks are below the application layer (like on TCP or HTTP), those are usually taken care of by our DNS and/or hosting provider (at least in the case of PaaS and FaaS). In this section, we'll look at application layer attacks, which can be separated into two buckets: expensive requests and a large number of requests. We want to guard against both.

First, guarding against expensive requests—requests that take up significant resources while the server processes them:

- **Safelisting:** If our API is private—only for use by our own client code—then we can safelist our queries. We'll send Apollo Graph Manager our client queries during a build step in the client repo(s), and then our server will check all incoming requests against the registered queries in Graph Manager and reject any unrecognized queries. If our API is public, however, we can't safelist, because we want third-party devs to be able to construct whatever queries they need.
- **Validate arguments:** Attackers can alter arguments to take up resources. For instance, if we have a `username` argument in our `signup` mutation, and then we save it to the database without checking the length, an attacker could provide a long string that takes up a gigabyte of hard drive space. Soon, our database would become full, which would prevent us from storing any further data.
- **Add a timeout:** If a request isn't done after N milliseconds, terminate it.
- **Limit depth:** One way to make a query expensive is to make it really deep—continuing to select connection fields (like `query { posts { comments { users { posts { comments { ...etc. }}}}}`). We can use the `graphql-depth-limit` library for this.
- **Limit complexity:** This is a more advanced technique than just limiting depth and involves assigning a complexity cost value to each field and limiting the total cost of a query. We can implement this using `graphql-validation-complexity`, or, if we want more flexibility, `graphql-cost-analysis`, which allows us to multiply costs by arguments or parent multipliers.

We can guard against a large number of requests by rate limiting. GitHub uses a combination of [rate limiting](#) and [cost analysis](#) for its public API—we can't make queries with a total cost of more than 5,000 points per hour. There's not yet an open-source library that does this. (If you write one, let us know so that we can link to it! And you may want to use a [leaky bucket algorithm](#) instead of a fixed window.) The `graphql-rate-limit-directive` library provides a directive that allows us to limit the number of times a particular field or object is selected within a certain time window.

- **Hide schema:** A common practice for private GraphQL APIs is disabling introspection in production. This is the default behavior of Apollo Server. While it doesn't guard against expensive operations, it makes it harder for an attacker to construct them, since they can't just open Playground and read through the schema.

In addition to blocking requests that are too complex or too frequent, we can reduce the amount of resources each request takes. For instance, instead of doing all the work needed during the request, in some cases we can send a response and then queue a job to be executed by a different server, clearing more room for our API server to handle more requests. Another example is caching—we can reduce the load on our database by using a cache, which we'll get to in the next section, [Performance > Caching](#).

Many of these techniques are implemented for us automatically when we use a backend-as-a-service like [Hasura](#).

## Performance

Background: [HTTP](#), [Latency](#), [Databases](#), [CDN](#)

Performance is mostly about speed—how quickly can the client receive a response. It's also about *load* (how much work a server is doing) since high load (caused by many concurrent requests) can result in either slower responses or no responses 🐢. *Capacity* is defined as either the load a server can handle before it fails to respond or before its

response speed decreases.

There are many places in the request-response cycle where we can improve speed or increase capacity. They all have different costs (in terms of development time, maintenance, and money) and different levels of improvement. An essential aspect of performance engineering is measurement. We need to know how long things take or how much load we can handle before we:

1. Decide we want to improve (performance / scalability is a common area of premature optimization).
2. Make improvements (so we can compare measurements before and after to determine how effective the change is).

We can determine our capacity with *load testing*, using `k6` with `easygraphql-load-tester` to make many simultaneous requests. We can measure server-side performance with Graph Manager like we did in the [Analytics](#) section: request rate and response time, as well as resolver timelines. Resolvers usually spend most of their time making database queries (which we'll examine in the next section, [Data fetching](#)), but if we wanted to look at exactly how long each one takes, we could do that as well (how we do that depends on which database we're using).

We also want to measure the response time from the client in order to spot:

- Longer times due to latency or limited bandwidth.
- Shorter times due to CDN or browser caching.

[Caching](#) has its own section, but here are a couple of other ways to improve speed measured from the client:

- Use an HTTP/2 server (like Node.js 10+).
- Use automatic persisted queries (APQ).

**HTTP/2:** Browsers limit the number of HTTP/1.1 connections to a single server, so if more than a certain number of requests (usually six) are made, the ones beyond six wait until the first six are completed. This drastically increases the time it takes the ones beyond six to complete. We can fix this by using HTTP/2, which can make multiple requests over a single connection.

**APQ:** When the client's requests include large queries and they're on a low-bandwidth connection, it can take a long time to send the request. Automatic persisted queries allow the client to send a hash of the query instead of the whole thing. It's enabled by default in Apollo Server and [with a link](#) on the client. The client creates a hash (a relatively small string) of the query and sends that to the server. The first time the server receives a hash, it doesn't recognize it and returns an error. Then the client replies with the full query and the hash, which the server saves. After that, whenever any client sends that hash, the server will recognize it and know which query to execute.

It's also possible to persist database queries (called *prepared statements* in SQL), in which the query is stored in the database and the API server just sends the query ID and arguments. This is done for us automatically when using [Hasura](#).

Before a request's processing reaches our resolvers, the GraphQL server library has to parse and validate the request. Then, during the execution phase, the library calls our resolvers. Different GraphQL servers do this process faster than others. For Node.js, the main improvement available is compiling queries to code, which `graphql-jit` does. It integrates with Apollo Server [like this](#). Another option for Python, Ruby, and Node is [Quiver](#).

## Data fetching

The largest server-side factor that contributes to the response time is how long resolvers take to return, and the majority of resolvers' runtime is usually taken up by fetching data. In this section, we'll cover the performance of data fetching in our resolvers.

Some of this section will apply to subscriptions. We also discussed scaling subscription servers in [Subscriptions in depth > Server architecture](#).

The three general speed factors, in order of importance:

1. How many data requests are made in series
2. How long the data source takes to get the data
3. Latency between our GraphQL server and the data source

Here *data source* means a source of data, like a database or an API—not an Apollo Server data source class.

Usually, we locate both our GraphQL server and our data sources in the same location, in which case #3 is very small (~0.2ms when inside the same AWS Availability Zone). However, when they're far apart—for instance when the data source is an external API hosted across the country—#3 can become a larger factor than #2.

Factor #2 depends on the type of data source and what data is being requested. For databases, usually the largest factor is if there is an index that covers the query—otherwise, the database has to search through all records in the table/collection, which takes much more time. Another large factor is whether data has to be read from disk—it's faster when the data is already in RAM. (MongoDB [recommends](#) having enough RAM to fit the *working set*—the indexes and data that are accessed frequently.)

Since different types of databases work differently, we may get faster results by using another database, in which case we might move or duplicate part or all of our data to the other database. For instance, Elasticsearch handles search queries more efficiently than our main database. We would duplicate all the data we wanted searchable from our main database to Elasticsearch, and then we would resolve all searches by querying Elasticsearch. Another type of query that is slow in many databases is one that skips a large number of results. This issue, which we talk about in the [Pagination section](#), is one reason to use [cursors](#).

Another factor that can improve database speed and reduce load is avoiding overfetching—instead of fetching all the fields (for instance `SELECT * FROM reviews`), we can fetch only the ones needed for the current query's selection set. If we use a library like [Join Monster](#) or a platform like [Hasura](#), this is done for us, as well as JOINS. Otherwise we can look at [info](#), the fourth resolver argument, to look up which fields to select.

A large area in which we can reduce load on a data source is sending fewer queries! One issue of basic implementations of GraphQL resolvers and ORMs is the *N+1 problem*. Consider this query:

```
query {
  post(id: "abc") {
    comments {
      id
      text
    }
  }
}
```

The N+1 problem is when our server does 1 query for the post document and then N comment queries—one for each ID in the `post.commentIds` array. There are actually two issues with this:

- The comment queries are done in parallel, but the post and the group of comment queries are done in series—the post is fetched before the comments. This is a significant hit to our GraphQL server's response time.
- When there are a lot of comments, there are a lot of comment queries, which is a high load on the server.

The second issue is fixed by `DataLoader`, which batches all the comment queries into a single query. To learn how to use `DataLoader`, see the [Custom data source](#) section. Also, if our data source is existing REST APIs, we can generate `DataLoader` code with Yelp's [dataloader-codegen](#) library.

To fix the first issue, we need the `post` resolver to fetch both the post and the comments at the same time. If we use [Join Monster](#) or [Hasura](#), this is done for us. If we use MongoDB, we have two options:

- Use a de-normalized structure in the posts collection, storing an array of comment objects inside each post document—then fetching the post will get the comments as well.



- Use the `info` resolver arg:
  - Store each comment with a `postId` field.
  - Look at `info` to see if `comments` is selected.
  - If it is selected, query for both the post and the comments at the same time.

```
const resolvers = {
  Query: {
    post: async (_, { id }, { dataSources }, info) => {
      const postPromise = dataSources.posts.findOneById(id)

      if (commentsIsSelected(info)) {
        const [post, comments] = await Promise.all([
          postPromise,
          dataSources.comments.findAllByPostId(id)
        ])
        post.comments = comments
        return post
      } else {
        return postPromise
      }
    }
  }
}
```

We can use this `info` technique with other databases as well as beyond the N+1 problem—there may be other queries we can initiate early. Viewing data in the `info` object can be simplified with the [graphql-parse-resolve-info](#) library.

## Caching

Wikipedia's [definition](#)) of a cache is “a hardware or software component that stores data so that future requests for that data can be served faster.” In addition to improving speed, caching also reduces load on the part of the system that originally provided the data that's being cached. For instance, a CDN caching an HTTP response reduces load on our server, which originally provided the response. And our `MongoDataSource` caching documents reduces load on our MongoDB database.

Here are the possible places for caches, starting in the client code that's requesting data, and ending with the database:

- **Client library:** GraphQL client libraries like Apollo Client [cache](#) response data from previous requests in memory.
- **Browser / Client OS:** Browsers, iOS, and Android cache HTTP responses based on the `Cache-Control` HTTP header.
- **CDN:** CDNs also cache HTTP responses based on `Cache-Control` (see [Background > CDN](#)).
- **Application server:**
  - Our GraphQL server can cache GraphQL responses in a caching database like Redis.
  - Our server's data source classes can [cache database responses](#) in Redis.
- **Database:** Our database has various levels of caching—in its software that uses RAM, in the operating system, and in the hard drives.

It's caches all the way down. —Yoav Weiss

Apollo Server will set the `Cache-Control` header for us as well as save the response to the cache. By default, however, it assumes we don't want data cached and doesn't do so. We have to tell it which fields and types we want cached and for how long. Then, if a response includes only those fields, it will set the header and save the response in the cache.

We can tell Apollo Server which fields and types we want cached with a *cache hint*. We can provide the hint in two ways:

- The `@cacheControl` schema directive
- Calling `info.cacheControl.setCacheHint()` in our resolvers

The first method we can use on both types and fields:

```
type Query {
  hello: String!
  reviews: [Review!]! @cacheControl(maxAge: 120)
  user(id: ID!): User
}

type Review @cacheControl(maxAge: 60) {
  id: ID!
  text: String!
  stars: Int
  commentCount: Int! @cacheControl(maxAge: 30)
}

type User @cacheControl(maxAge: 600) {
  id: ID!
  firstName: String!
  reviews: [Review!]!
}
```

`maxAge` is in seconds. The lowest `maxAge` is used. For instance, `Review.commentCount` has a `maxAge` of 30, so the response to the below query would be cached for 30 seconds:

```
query {
  user(id: "1") {
    reviews {
      text
      stars
      commentCount
    }
  }
}
```

Whereas this would be cached for 60:

```
query {
  user(id: "1") {
    reviews {
      text
      stars
    }
  }
}
```

Similarly, if we didn't select `User.reviews`, the hint on `User` would be used, and the below query would be cached for 10 minutes:

```
query {
  user(id: "1") {
    firstName
  }
}
```

Field cache hints override type hints, so for the below query, `Query.reviews`'s `maxAge: 120` would be used instead of `Review`'s `maxAge: 60`:

```
query {
  reviews {
    text
    stars
  }
}
```

```
}
}
```

Finally, neither of the below queries would be cached, as `query.hello` doesn't have a hint:

```
query {
  hello
}
```

```
query {
  reviews {
    text
    stars
  }
  hello
}
```

There's one more directive argument: `scope`. It's `PUBLIC` by default, and the other value is `PRIVATE`:

```
type Query {
  me: User! @cacheControl(maxAge: 300, scope: PRIVATE)
}
```

Apollo would set the response header to `Cache-Control: max-age=300, private`. Including `private` means that the response should only be stored in a browser's cache, not a CDN. Because if a CDN stored `Query.me` (the current user's account), other clients who made the query would get access to the first user's account data.

Some advanced CDNs like [Cloudflare](#) actually support caching private responses by matching them to a single user with an authentication token. Similarly, Apollo Server supports caching responses through a function that returns a session ID or any unique string associated with a user—in the below code, we use the JWT:

```
import responseCachePlugin from 'apollo-server-plugin-response-cache';

const server = new ApolloServer({
  ...,
  plugins: [
    responseCachePlugin({
      sessionId: requestContext =>
        requestContext.request.http.headers.get('authorization') || null,
    })
  ]
});
```

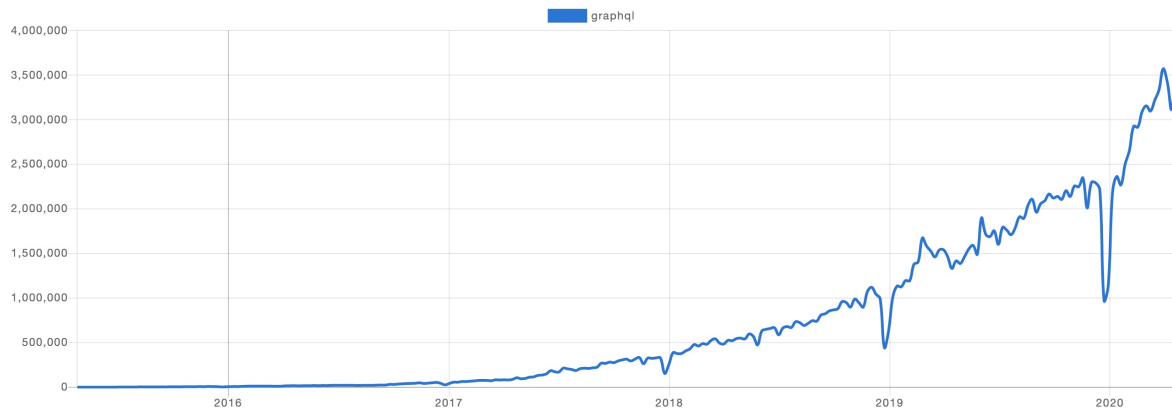
If Apollo caches a response with scope `PRIVATE`, it will also save the session ID. If the same request arrives later, and the same session ID is returned from this function, Apollo will use the cached response.

One issue with CDN caching is that many CDNs only cache GET requests, and GraphQL requests are usually made via POSTs. Apollo Server supports GET requests, and clients can switch to using them, but GET requests have the query in the URL, and sometimes queries are too long to fit in a URL. However, we can use automatic persisted queries (discussed [earlier](#)), which results in clients using cacheable GET requests with short URLs, regardless of the query length.

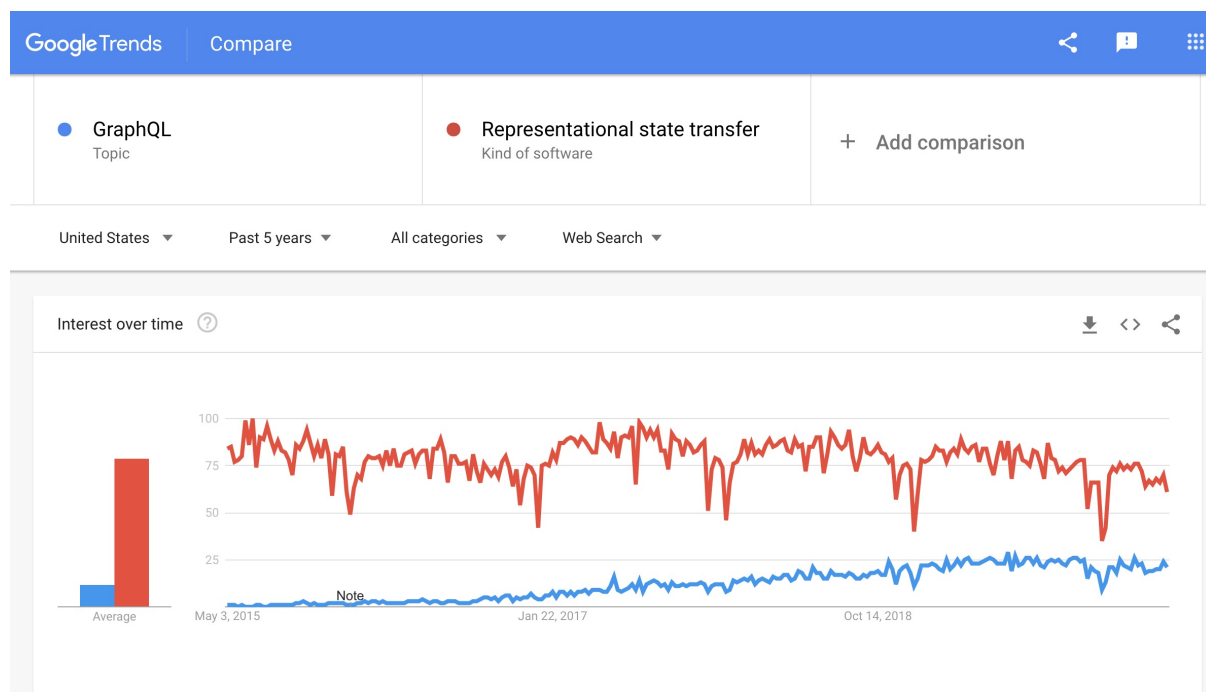
## Future

The largest change to GraphQL-land in the coming years will be its size! The [S curve](#) of GraphQL adoption is currently in the exponential phase (*seemingly* exponential—technically, it's logistic). Here's a graph of the [graphql](#) package's weekly npm downloads over the first 5 years:

Downloads in past 5 Years ▾



And it doesn't even include client packages like `apollo-client`, which are also growing. At the time of writing, `apollo-client` has 1.7M weekly downloads. There's also a lot of room left to grow—according to Google Trends, REST still has 3x the interest that GraphQL has:



As adoption grows, more resources will be put into GraphQL libraries, tools, and services. The existing ones will improve, and new ones will be created.

- Apollo Server's [roadmap](#) lists near-term future work, including:
  - Adding subscription support to Apollo Federation.
  - Adding `@defer` and `@stream` directives.
  - Invalidation of whole-query cache through cache tags with CDN integration.
  - Building a "graph" caching layer for the gateway.
- Apollo Client also has a [roadmap](#) as well.
- For some futuristic-seeming services and tooling, check out [this video](#) from the creator of [OneGraph](#) (a GraphQL API that combines many different companies' APIs).
- An exciting area in which we're looking forward to growth is full-stack GraphQL frameworks—the Ruby on Rails of GraphQL, Node, and React. Our current favorites are [RedwoodJS](#) (a new project based on serverless and Prisma) and [Vulcan.js](#) (a mature project based on Meteor and MongoDB).

There will also be changes to the language itself. In 2018, Facebook transferred the GraphQL project (which includes the spec, the `graphql-js` reference implementation, GraphiQL, and DataLoader) to a new Linux Foundation called the [GraphQL Foundation](#). Anyone can discuss or propose changes to the specification in its GitHub repo, [graphql/graphql-spec](#), or in the [GraphQL Working Group](#), a monthly virtual meeting of maintainers.

Changes to the spec go through an [RFC process](#), and the current proposals are [listed here](#). A few of them are:

- The `@defer` and `@stream` query directives we mentioned on the Apollo Server roadmap. Adding `@defer` to a field tells the server they can initially return `null` and later fill in the data. Adding the `@stream` directive to a field with a list type means the server can send part of the list initially, and further parts of the list later. These directives address the fact that currently the server only sends a single response, which means it has to wait for all data to arrive from its data sources. And that means the response time is limited by the slowest source. With `@defer` and `@stream`, the client can get some of the data sooner.
- The `@live` query directive, which means: “send me the current value of this field, and then send me the updated value whenever it changes.”
- The Input Union—creating a union type that can be used for arguments. The [proposal](#) (a.k.a. RFC) is a long document that starts with:

#### RFC: GraphQL Input Union

The addition of an Input Union type has been discussed in the GraphQL community for many years now. The value of this feature has largely been agreed upon, but the implementation has not.

This document attempts to bring together all the various solutions and perspectives that have been discussed with the goal of reaching a shared understanding of the problem space.

From that shared understanding, the GraphQL Working Group aims to reach a consensus on how to address the proposal.

There are also specifications in GraphQL-land other than the GraphQL spec, including the [Relay Cursor Connections](#) spec, the [Relay server](#) spec, and the in-development [GraphQL over HTTP](#) spec.

You can contribute to the future of GraphQL by:

- Building things with it!
- Contributing to GraphQL libraries and tools.
- Getting involved with the spec and foundation.
- Spreading the word.

Speaking of spreading the word, if you'd like to recommend the Guide to a friend or co-worker, we'd appreciate it 🙏. <https://graphql.guide>. And we'd value any feedback you may have on the book via [GitHub issues](#) or PRs.

To learn more about GraphQL, we recommend:

- Books:
  - [Production Ready GraphQL](#): An in-depth discussion of production topics.
  - [Advanced GraphQL with Apollo & React](#): A large tutorial-style book based on Apollo Federation and React.
- Course: [Fullstack Advanced React & GraphQL](#)
- Reading [the spec](#)