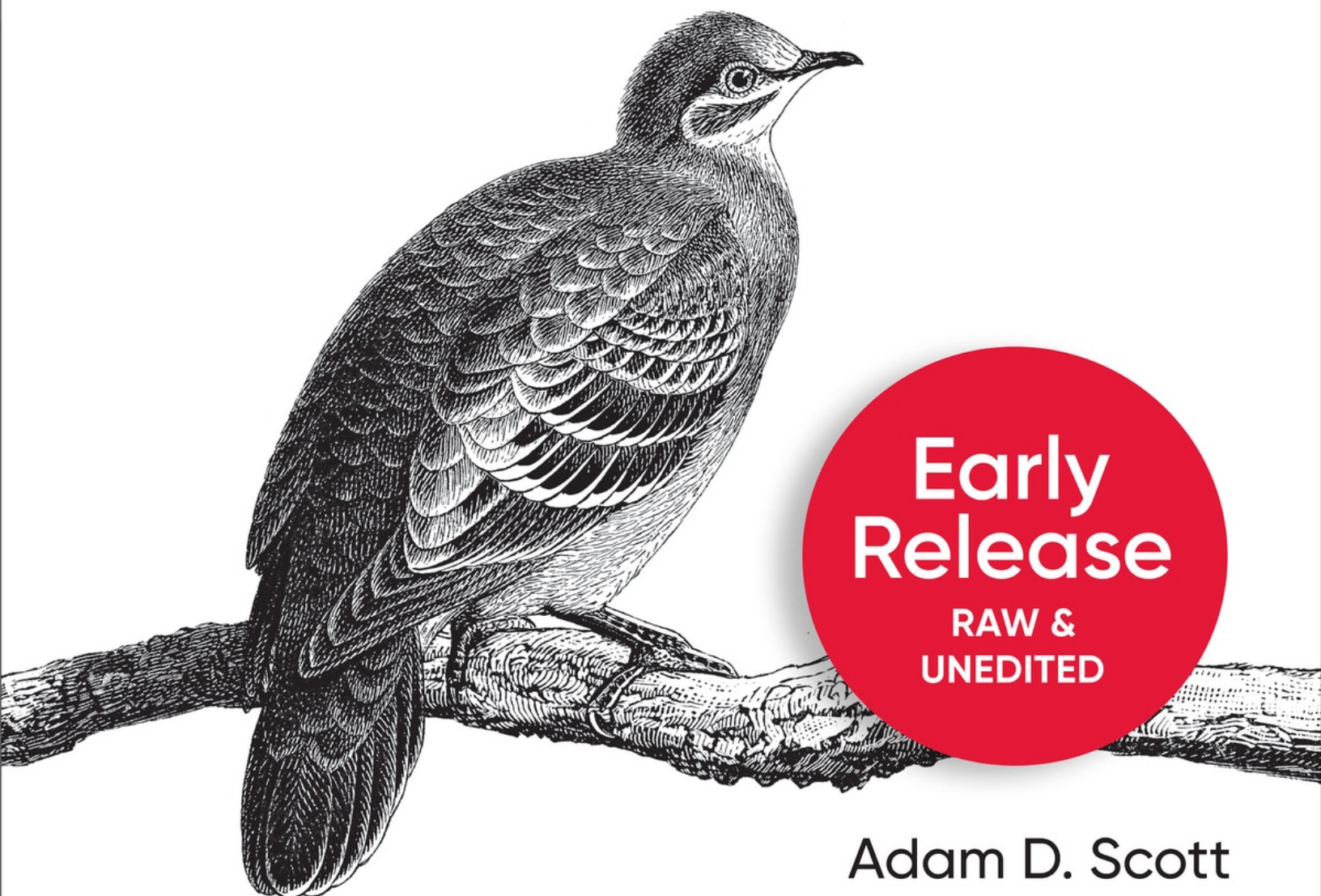


O'REILLY®

JavaScript Everywhere

Building Cross-platform Applications with
GraphQL, React, React Native, and Electron



Early
Release

RAW &
UNEDITED

Adam D. Scott

1. Preface
 - a. Conventions Used in This Book
 - b. O'Reilly Online Learning
 - c. How to Contact Us
2. 1. Our Development Environment
 - a. Your text editor
 - b. The terminal
 - i. Using VSCode
 - ii. Using the built in terminal
 - iii. Navigating the file system
 - c. Command Line Tools and Homebrew (Mac Only)
 - d. Node.js and npm
 - i. Installing Node.js and npm for macOS
 - ii. Installing Node.js and npm for Windows
 - e. MongoDB
 - i. Installing MongoDB for macOS
 - f. Installing MongoDB for Windows
 - g. Git
 - h. Expo
 - i. Prettier
 - j. ESLint
 - k. Making things look nice

1. Conclusion
3. 2. API Introduction
 - a. What we're building
 - b. How we're going to build this
 - c. Getting started
 - d. Conclusion
4. 3. A Web Application with Node and Express
 - a. Hello World
 - b. Nodemon
 - c. Extending our port options
 - d. Conclusion
5. 4. Our First GraphQL API
 - a. Turning our server into an API(sort of)
 - b. GraphQL Basics
 - i. Schemas
 - ii. Resolvers
 - c. Adapting our API
 - d. Conclusion

JavaScript Everywhere

Building Cross-platform Applications with
GraphQL, React, React Native, and Electron

Adam D. Scott



JavaScript Everywhere

by Adam D. Scott

Copyright © 2020 Adam D. Scott. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Jennifer Pollock and Corbin Collins

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

February 2020: First Edition

Revision History for the Early Release

- 2019-04-18: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492046981> for release

details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *JavaScript Everywhere*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04691-2

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

O'Reilly Online Learning

NOTE

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

To comment or ask technical questions about this book, please send an email to [*bookquestions@oreilly.com*](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at [*http://www.oreilly.com*](http://www.oreilly.com).

Find us on Facebook: [*http://facebook.com/oreilly*](http://facebook.com/oreilly)

Follow us on Twitter: [*http://twitter.com/oreillymedia*](http://twitter.com/oreillymedia)

Watch us on YouTube: [*http://www.youtube.com/oreillymedia*](http://www.youtube.com/oreillymedia)

Chapter 1. Our Development Environment

John Wooden, the late coach of the UCLA men's basketball team is one of the most successful coaches of all time, winning 10 national championships in a twelve year period. His teams consisted of top recruits, including hall-of-fame players such as Lew Alcindor (Kareem Abdul-Jabbar) and Bill Walton. On the first day of practice, John Wooden would sit down with each of his new recruits, players who had been the best in the country in high school, and teach them to put on their socks properly. When asked about this, Wooden stated that "it's the little details that make the big things come about."

Chefs use the term *mise en place*, meaning "everything in its place," to describe the practice of preparing the tools and ingredients required for the menu prior to cooking. This preparation enables the kitchen's cooks to successfully prepare meals during busy rushes, as the small details have already been considered. Much like Coach Wooden's players and chefs preparing for a dinner rush, it is worth dedicating time to setting up our development environment.

A useful development environment does not require expensive software or top of the line hardware. In fact, I'd encourage you to start simple, use open source software, and grow your tools with you. Though a runner prefers a specific brand of sneakers and a carpenter may always reach for her favorite hammer, it took time and experience to establish these preferences. Experiment with tools, observe others, and over time you will

create the environment that works best for you.

In this chapter we'll install a text editor, Node.js, Git, MongoDB, and several helpful JavaScript packages as well as locate our terminal application. It's possible that you already have a development environment that works well for you, however we will also be installing several required tools that will be used throughout the book. If you're like me and typically skip over the instruction manual, I'd still encourage you to read through this guide.

If you find yourself stuck at any point, please reach out to the JavaScript Everywhere community, via our Spectrum channel at spectrum.chat/jseverywhere.

Your text editor

Text editors are a lot like pants. We all need them, but our preferences may vary wildly. Some like simple, well-constructed, and timeless. Some prefer the flashy paisley pattern. There's no wrong decision and you should use whatever makes you most comfortable.

If you don't already have a favorite, I highly recommend Visual Studio(VS) Code. It's an open source editor that is available for Mac, Windows, and Linux. Additionally, it offers built-in features to simplify development and is easily modified with community extensions. It's even built using JavaScript!

The terminal

If you're using Visual Studio Code, it comes with an integrated terminal.

For most development tasks, this may be all you need. Personally, I find using a dedicated terminal client preferable as I find it easier to manage multiple tabs and use more dedicated window space on my machine. I'd suggest trying both out and find what works best for you.

Using VSCode

To access the terminal in VSCode, click **View > Integrated Terminal**. This will present you with a terminal window. The prompt will be present in the same directory as the current project.

Using the built in terminal

All operating systems come with a built in terminal application and this is a great place to get started. On OS X it is called, fittingly enough, Terminal. In Windows, the program is PowerShell. The name of the terminal for Linux distributions may vary, but often include "Terminal."

Navigating the file system

Once you've found your terminal, the most critical ability you will need is the ability to navigate the file system. This can be done using the `cd` command, which stands for "change directory."

COMMAND LINE PROMPTS

When looking at terminal instructions they will often include a `$` or `>` at the start of the line. These are used to designate the prompt and should not be copied. In this book, I'll be designating the terminal prompt with a dollar sign (`$`). When entering instructions into your terminal application, do not copy the `$`.

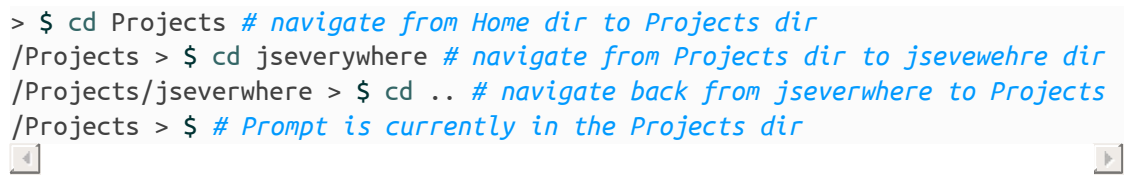
When we open our Terminal application, we'll be presented with a blinking cursor prompt. By default, we are in our computer's home directory. If you haven't already, I'd recommend making a **Projects** folder that is a sub-directory within your home directory. This folder can house all of your development projects. To navigate into that folder you would type:

A screenshot of a terminal window with a light gray background. The prompt '\$ cd Projects' is entered in a monospaced font. On the left side of the terminal bar, there is a small square button with a left-pointing arrow, and on the right side, there is a small square button with a right-pointing arrow.

```
$ cd Projects
```

Now let's say that we can have a folder called **jseverywhere** in our **Projects** directory. We can type `cd jseverywhere` from the **Projects** directory to navigate into there. To navigate backwards a directory (in this case, to **Projects**), we would type `cd ..`

All together, this would look something like:

A screenshot of a terminal window showing a sequence of directory navigation commands. The text is color-coded: the prompt and directory names are in blue, and the comments are in red. The commands are: '\$ cd Projects', '# navigate from Home dir to Projects dir', '/Projects > \$ cd jseverywhere', '# navigate from Projects dir to jsevewehre dir', '/Projects/jseverwhere > \$ cd ..', '# navigate back from jseverwhere to Projects', and '/Projects > \$ # Prompt is currently in the Projects dir'. On the left side of the terminal bar, there is a small square button with a left-pointing arrow, and on the right side, there is a small square button with a right-pointing arrow.

```
> $ cd Projects # navigate from Home dir to Projects dir
/Projects > $ cd jseverywhere # navigate from Projects dir to jsevewehre dir
/Projects/jseverwhere > $ cd .. # navigate back from jseverwhere to Projects
/Projects > $ # Prompt is currently in the Projects dir
```

If this is new to you, spend some time navigating through your files until you're comfortable. I've found that file system issues are a common tripping point for budding developers. Having a solid grasp of this will provide you with a solid basis for establishing your workflows.

Command Line Tools and Homebrew (Mac Only)

Certain command line utilities are only available to macOS users once Xcode is installed. You can jump through this hoop, without installing

Xcode by installing `xcode-select` via your terminal. To do so, run the following command and click through the install prompts:

```
$ xcode-select --install
```

Homebrew is a package manager for macOS. It makes installing development dependencies, like programming languages and databases as simple as running a command line prompt. If you use a Mac, it will dramatically simplify your development environment. To install Homebrew, either head over to brew.sh to copy and paste the install command, or type the following:

```
$ /usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Node.js and npm

Node.js is “a JavaScript runtime, built on Chrome’s V8 JavaScript Engine.” In practical terms this means that Node is a platform that allows developers to write JavaScript outside of a browser environment. Node.js enables us to write server-side applications in JavaScript. Bundled along with Node.js, comes npm, the default package manager for Node.js. npm enables us to install thousands of libraries and JavaScript tools within our projects.

MANAGING NODE.JS VERSIONS

If you plan on managing a large number of Node projects you may find that you also need to manage multiple versions of Node on your machine. If that’s the case, I recommend using [nvm](#) to install Node. nvm is a script that enables you to manage multiple active Node versions. For Windows users, I recommend [nvm-windows](#). I won’t be covering Node versioning, but it is a helpful tool. If this is your first time

working with Node, I recommend proceeding with the following instructions for your system.

Installing Node.js and npm for macOS

macOS users can install Node.js and npm using Homebrew. To install Node.js, type the following command into your terminal:

```
$ brew update  
$ brew install node@10.15
```

With Node installed, open your terminal application to verify it is working.

```
$ node --version  
## Expected output v10.15.3  
$ npm --version  
## Expected output 6.2.0
```

If you see a version number after typing those commands, congratulations you've successfully installed Node and npm for macOS!

Installing Node.js and npm for Windows

For Windows, the most straightforward way to install Node.js is to visit [Node.org](https://node.org) and download the installer for your operating system.

First, visit [Node.org](https://node.org) and install the LTS version (10.15.3 at the time of writing), following the installation steps for your operating system. With Node installed, open your terminal application to verify it is working.

```
$ node --version  
## Expected output v10.15.3
```

```
$ npm --version  
## Expected output 6.2.0
```

If you see a version number after typing those commands, congratulations you've successfully installed Node and npm for Windows!

MongoDB

MongoDB is the database that we will be using in the development of our API. Mongo is a popular choice when working with Node.js, because it treats our data as JSON documents. This means that it's comfortable for JavaScript developers to work with from the get-go.

Installing MongoDB for macOS

To install MongoDB for macOS, first install with Homebrew:

```
$ brew update  
$ brew install mongodb
```

Now, we will create a directory to which Mongo will write our data and give it the proper permissions. Within your terminal, run the following commands:

```
$ cd  
$ sudo mkdir -p /data/db  
# If prompted, enter your password when prompted and press `return`  
sudo chown -R `id -un` /data/db  
# If prompted, enter your password when prompted and press `return`
```

To verify that Mongo has installed and start the Mongo Daemon, type `mongod` into your terminal. This should start the Mongo server. To stop the server and exit the Mongo process, hold `ctrl+c`.

Installing MongoDB for Windows

To install MongoDB for Windows, first download the installer from the MongoDB Download Center at <https://www.mongodb.com/download-center/community>. Once the file has downloaded, run the installer.

Once installation is complete, we will need to create a directory in which Mongo will write our data. Within your terminal, run the following commands:

```
$ cd C:\
$ md "\\data\db"
```

To verify that Mongo has installed and start the Mongo Daemon, type `C:\mongodb\bin\mongod.exe` into your terminal. This should start the Mongo server.

TROUBLE INSTALLING MONGO ON WINDOWS?

Note that, the location of `mongod.exe` may be different depending on your system preferences. If that prompt doesn't work, you may need to track it down. The MongoDB installation documentation offers a more extensive [Windows installation guide](#).

Git

Git is the most popular version control software, allowing you to do things like copy code repositories, merge code with others, and create branches of your own code that do not impact one another. Git will be helpful for “cloning” this book’s sample code repositories, meaning it will allow you to directly copy a folder of sample code. Depending on your operating

system, Git may already be installed. Type the following into your Terminal window:

```
$ git --version
```

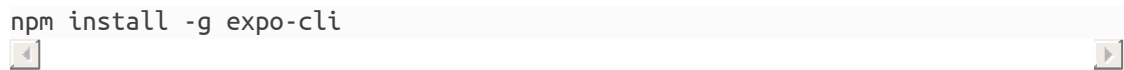


If a number is returned, congrats you're all set! If not, visit git-scm.com to install Git, or use Homebrew for macOS. Once you've completed the installation steps, once again type `git --version` into your terminal to verify that it has worked.

Expo

Expo is a toolchain that simplifies the bootstrapping and development iOS and Android projects with React Native. I recommend installing the Expo command line tool as well as the Expo app for iOS or Android. We'll cover this in more detail in the mobile application portion of the book, but if you're interested in getting a head start visit expo.io to learn more. To install the command line tools, type the following into your terminal:

```
npm install -g expo-cli
```



Prettier

Prettier is a code formatting tool with support for a number of languages including JavaScript, HTML, CSS, GraphQL, and Markdown. It makes it easy to follow basic formatting rules, meaning that when you run the Prettier command, your code is automatically formatted to follow a standard set of best practices. Even better, you can configure your editor to do this automatically every time you save a file. This means that you'll never again have a project with things like inconsistent spaces and mixed

quotes.

I recommend installing Prettier globally on your machine and configuring a plugin for your editor. To install Prettier globally, go to your command line and type

```
npm install --global prettier
```

Once you've installed Prettier, visit [Prettier.io](https://prettier.io) to find the plugin for your text editor. With the editor plugin installed, I recommend adding the following settings, within your editor's settings file:

```
"editor.formatOnSave": true,  
"prettier.requireConfig": true
```

These settings will automatically format files on save, whenever a `.prettierrc` configuration file is within the project. The `.prettierrc` file specifies specific options for prettier to follow. Now whenever that file is present, your editor will automatically reformat your code to meet the conventions of the project. Each project within this book will include a `.prettierrc` file.

ESLint

ESLint is a code linter for JavaScript. A linter differs from a formatter, such as Prettier, in that a linter also checks for code quality rules, such as unused variables, infinite loops, and unreachable code that falls after a return. Much like Prettier, I recommend installing the ESLint plugin for your favorite text editor. This will alert you to errors in real time as you write your code. You can find a list of editor plugins on the ESLint Website, at eslint.org/docs/user-guide/integrations.

Similar to Prettier, projects can specify the ESLint rules they would like to follow within an `.eslintrc` file. This allows project maintainers fine grained control over their code preferences and a means to automatically enforce coding standards. Each of the projects within this book will include a helpful, but permissive set of ESLint rules, aimed at helping you to avoid common pitfalls.

Making things look nice

This is optional, but I've found that I enjoy programming just a bit more when I find my setup aesthetically pleasing. I can't help it, I have a degree in the arts. Take some time and test out different color themes and typefaces. Personally, I've grown to love the [Dracula Theme](#), which is a color theme available for nearly every text editor and terminal, along with Adobe's [Source Code Pro](#) typeface.

Conclusion

In this chapter we've set up a working and flexible JavaScript development environment on our computer. One of the great joys of programming is personalizing your environment. I encourage you to experiment with the themes, colors, and tools that you use to make this environment your own. In the next section of the book, we will put this environment to work by developing our API application.

Chapter 2. API Introduction

Picture yourself seated in a booth at a small, local restaurant where you've decided that you'd like to order a sandwich. The server writes your order on a slip of paper and passes that paper to the cook. The cook reads the order, takes individual ingredients to build the sandwich, and passes the sandwich to the server. The server then brings the sandwich to you to eat. If you decide, you'd then like some desert the process repeats.

An Application Programming Interface (API) is a set of specifications that allow one computer program to interact with another. A Web API works in much the same way that we ordered our sandwich. A client requests some data, that data travels to a web server application over the HyperTextTransfer Protocol (HTTP), the web server application takes the requests and processes data, the data is then sent to the client over HTTP.

In this chapter we'll explore the broad topic of Web APIs and get started with our development by cloning the starter API project to our local machine. Before we do that, however, let's explore the requirements of the application that we'll be building.

What we're building

Throughout the book we'll be building a social note application, called Notedly. Users will be able to create an account, write notes in markdown, edit their notes, view a feed of other user's notes, and "favorite" the notes of other users. In this portion of the book, we'll be developing the API to

support this application.

The requirements of our API are as follows: - Users will be able to create notes as well as read, update, and delete the notes they've created. - Users will be able to view a feed of notes created by other users, and read individual notes created by others, though they will not be able to update or delete them. - Users will be able to create an account, login, and logout. - Users will be able to retrieve their profile information as well as the public profile information of other users. - Users will be able to favorite the notes of other users as well as retrieve a list of their favorites.

Though this sounds like a lot, we'll be breaking it down into small chunks throughout this portion of the book. Once you've learned to perform these types of interactions you'll be able to apply them to building all sorts of APIs.

How we're going to build this

To build our API we'll be using the GraphQL API query language. GraphQL is an open source specification, first developed at Facebook in 2012. The advantage of GraphQL is that it allows the client to request precisely the data it needs, dramatically simplifying and limiting the number of requests. This also provides a distinct performance advantage when sending data to mobile clients, as we only need to send the data that the client needs. We'll be exploring how to write, develop, and consume GraphQL APIs throughout much of the book.

WHAT ABOUT REST?

If you're familiar with Web API terminology, you've likely heard of REST (Representational State Transfer) APIs. The REST architecture has been (and

continues to be) the dominant format for APIs. These APIs differ from GraphQL in structure, relying upon the URL structure and query parameters to make requests to a server. While REST remains relevant, the simplicity of GraphQL, the robustness of tooling around GraphQL, and the potential performance gains of sending limited data over the wire, make GraphQL my preference for modern platforms.

Getting started

Before we can start development, we will need to make a copy of the project starter files to our machine. The project's source code is located at github.com/javascripteverywhere/api. To clone the code to our local machine, open the terminal, navigate to the directory where you keep your projects, and `git clone` the project repository. It may also be helpful to create a `notedly` directory, to keep our project's code organized:

```
$ cd Projects
$ mkdir notedly && cd notedly
$ git clone git@github.com:javascripteverywhere/api.git
$ cd api
```

The code is structured as follows:

- `/src`: This is the directory where you should perform your development as you follow along with the book.
- `/solutions`: This directory contains the solutions for each chapter. If you get stuck, these are available for you to consult.
- `/final` This directory contains the final working project.

Now that we have the code on our local machine, we will need to make a copy of the project's `.env` file. This file is a place to keep the project secrets, such as the database URL, client ID's, and passwords. Because of this we never want to check it into source control. You'll need your own

copy of the `.env` file. To do this, type the following into your terminal, from the `api` directory:

```
cp .env.example .env
```

We should now see a `.env` file in the directory. We don't yet need to do anything with this file, but will be adding information to it as we progress through the development of our API back-end. The `.gitignore` file included with the project will ensure that we do not inadvertently commit our `.env` file.

HELP, I DON'T SEE THE .ENV FILE!

By default, operating systems hide files that start with a period, as these are typically used by the system, not end users. If you don't see the `.env` file, try opening the directory in your text editor. The file should be visible in the file explore of your editor. Alternately, typing `ls -a` into your terminal window will list the files in the current working directory.

Conclusion

APIs provide an interface for data to flow from a database to applications. In doing so, they are the backbone of modern applications. By using GraphQL we are able to quickly develop modern, scalable API-based applications. In the next chapter we'll begin our API development by building a web server, using Node.js and Express.

Chapter 3. A Web Application with Node and Express

Before implementing our API, we're first going to build a basic server-side web application. This application will act as the basis for the back-end of our API. We'll be using the Express framework. Express is a “minimalist web framework for Node.js,” meaning that it does not ship with a lot of features, but is highly configurable. We'll be using Express as the backbone of our API, but Express can also be used independently to build fully featured web applications.

User interfaces, such as web sites and mobile applications communicate with web servers when they need to access data. This data could be anything from the HTML required to render a page in a web browser to the results of a user's search. The client interface communicates with the server using the HyperTextTransfer Protocol (HTTP). The data request is sent from the client via HTTP to the web application that is running on our server. The web application then processes the request and returns the data to the client, again over HTTP.

In this chapter we'll build a small server-side web application, which will be the basis for our API. To do this, we'll use the Express.js framework to build a basic web application that sends a basic request.

Hello World

Now that we understand the basics of server-side web applications, let's

jump in. Within the `src` directory of our `api` project, create a file named `index.js` and add the following:

```
const express = require('express');
const app = express();

app.get('/', (req, res) => res.send('Hello World'));

app.listen(4000, () => console.log('Listening on port 4000!'));
```

In this example, we are first requiring the `express` dependency and creating the `app` object, using the imported `express` module. We then use the `app` object's `get` method to instruct our application to send a response of “Hello World” when a user accesses the `/` URL. Lastly, we are instructing our application to run on port 4000. This will allow us to view the application locally at the URL of `http://localhost:4000`.

Now to run our application, we type `node src/index.js` in our terminal. After doing so, we should see a log in our terminal that reads `Listening on port 4000!`. If that's the case, open a browser window to `http://localhost:4000` and we should see the following:

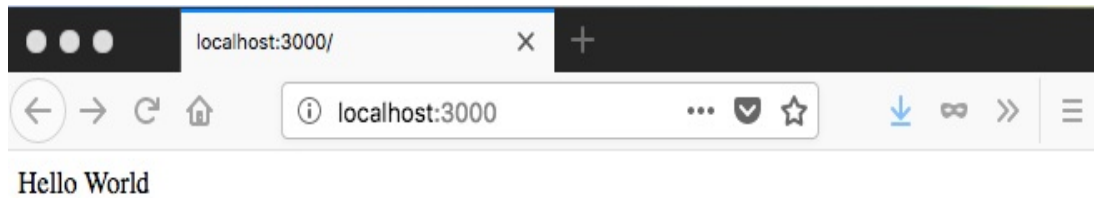


Figure 3-1. The results of our Hello World server code in the browser

Nodemon

Now, let's say that the output of this example doesn't properly express our excitement. We may want to change our code so that it adds an exclamation mark to our response. Let's go ahead and do that, changing the `res.send` value to `read: Hello World!!!`. The full line should now be:

```
app.get('/', (req, res) => res.send('Hello World!!!'));
```

If we go to our web browser and refresh the page, you'll notice that the output hasn't changed. This is because any changes we make to our web server require us to restart it. To do so, switch back to your terminal and type `CTRL + C` to stop the server. Now restart it by again typing `node index.js`. If we then navigate back to our browser and refresh the page we should see our updated response.

As you can imagine, stopping and restarting our server for every change we make can quickly become tedious. Thankfully, we can use the Node package `nodemon` to automatically restart our server on changes. If we take a look at our project's `package.json` we will see a `dev` command within the `scripts` object, that instructed `nodemon` to watch our `index.js` file:

```
"scripts": {  
  ...  
  "dev": "nodemon src/index.js"  
  ...  
}
```

NOTE

There are a handful of other helper commands within the `scripts` object. We will be exploring those in future chapters.

Now, when we want to start our application from the terminal we type:

```
npm run dev
```

Navigating to our browser and refreshing the page, we'll observe that things are working as before. To confirm that `nodemon` automatically restarts our server, let's once again update our `res.send()` value so that it reads:

```
res.send('Hello Web Server!!!')
```

Now, we should be able to refresh the page in our browser and see our update without manually restarting the server.

Extending our port options

Currently our application is served on port 4000. This works great for local development, but we will need the flexibility to set this to a different port number when working with a deployment environment. Let's take the steps to update this now. We'll start by adding a port variable.

```
const port = process.env.PORT || 4000;
```

This change will allow us to dynamically set the port in the Node environment, but fall back to port 4000 when no port is specified. Now let's adjust our `app.listen` code to work with this change and use a template literal to log the correct port:

```
app.listen(port, () =>
  console.log(
    `Server running at http://localhost:${port}`
  )
);
```

Our final code should now read:

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

const app = express();
const port = process.env.PORT || 4000;

app.get('/', (req, res) => res.send('Hello World!!!'));

app.listen(port, () =>
  console.log(
    `Server running at http://localhost:${port}`
  )
);
```

With this, we now have the basics of our web server code up and running. To test that everything is working, verify that no errors are present in your console and re-load your web browser at `http://localhost:4000`.

Conclusion

Server-side web applications are the foundation of API development. In this chapter, we built a basic web application using the Express.js framework. When developing Node-based web applications there is a wide array of frameworks and tools to choose from. Express.js is a great choice due to its flexibility, community support, and maturity as a project. In the next chapter, we'll turn our web application into an API.

Chapter 4. Our First GraphQL API

Presumably, if you are reading this, you are a human being. As a human being you have a number of interests and passions. You also have family members, friends, acquaintances, classmates, and colleagues. Each of those people also have their own social relationships, interests, and passions. Some of these relationships and interests overlap, while others do not. All together, for each of us, it builds a connected graph of the people in our lives.

These types of interconnected data are exactly the challenge that GraphQL initially set out to solve in API development. By writing a GraphQL API we are able to efficiently connect data, which reduces the complexity and number of requests while allowing us to serve a client precisely the data they need.

Does that all sound like a bit of overkill for a notes application? Perhaps it does, but as you'll see, the tools and techniques provided by the GraphQL JavaScript ecosystem both enable and simplify all types of API development.

In this chapter we'll build a GraphQL API, using the `apollo-server-express` package. To do so, we'll explore fundamental GraphQL topics, write a GraphQL schema, develop code to resolve our schema functions, and test our API using the GraphQL Playground user interface.

Turning our server into an API(sort of)

Let's begin our API development by turning our Express server into a GraphQL server using the `apollo-server-express` package. Apollo Server is an open-source GraphQL server library that works with a large number of Node.js server frameworks, including Express, Connect, Hapi, and Koa. It enables us to serve data as a GraphQL API from a Node.js application and also provides helpful tooling such as the GraphQL Playground, a visual helper for working with our API in development.

To write our API we'll be modifying the web application code within that we wrote in the previous chapter. Let's start by including the `apollo-server-express` package. Add the following to the top of your `src/index.js` file:

```
const { ApolloServer, gql } = require('apollo-server-express');
```

Now that we've imported `apollo-server`, we'll set up a basic GraphQL application. GraphQL applications consist of two primary components: a schema of type definitions and resolvers, which resolve the queries and mutations performed against the data. If that all sounds like nonsense, that's OK. We'll implement a "Hello World" API response and will further explore these GraphQL topics throughout the development of our API

To begin, let's construct a basic schema, which we will store in a variable called `typeDefs`. This schema will describe a single Query with the name of `hello` which will return a string:

```
// Construct a schema, using GraphQL schema language  
const typeDefs = gql`  
  type Query {  
    hello: String
```



```
}  
;  
}
```

Now that we've set up our schema, we can add a resolver that will return a value to the user. This will be a simple function that returns the string "Hello world!":

```
// Provide resolver functions for our schema fields  
const resolvers = {  
  Query: {  
    hello: () => 'Hello world!'  
  }  
};
```

Lastly, we'll integrate Apollo Server, to serve our GraphQL API. To do so, we'll add some Apollo Server specific settings and middleware and update our `app.listen` code:

```
// Apollo Server setup  
const server = new ApolloServer({ typeDefs, resolvers });  
  
// Apply the Apollo GraphQL middleware and set the path to /api  
server.applyMiddleware({ app, path: '/api' });  
  
app.listen({ port }, () =>  
  console.log(  
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`  
  )  
);
```

Putting it all together, our `src/index.js` file should now look like this:

```
const express = require('express');  
const { ApolloServer, gql } = require('apollo-server-express');  
  
// Run the server on a port specified in our .env file or port 4000
```

```

const port = process.env.PORT || 4000;

// Construct a schema, using GraphQL's schema language
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

// Provide resolver functions for our schema fields
const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};

const app = express();

// Apollo Server setup
const server = new ApolloServer({ typeDefs, resolvers });

// Apply the Apollo GraphQL middleware and set the path to /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);

```

If we've left our nodemon process running, we can head straight to the browser otherwise we must type `npm run dev` within our terminal application to start the server. We can then visit <http://localhost:4000/api> where we're greeted with the GraphQL Playground. This web app, which comes bundled with Apollo Server, is one of the great benefits of working with GraphQL. From here, we are able to run GraphQL queries and mutations and see the results. We are also able to click the Schema tab to access automatically created documentation for the API.

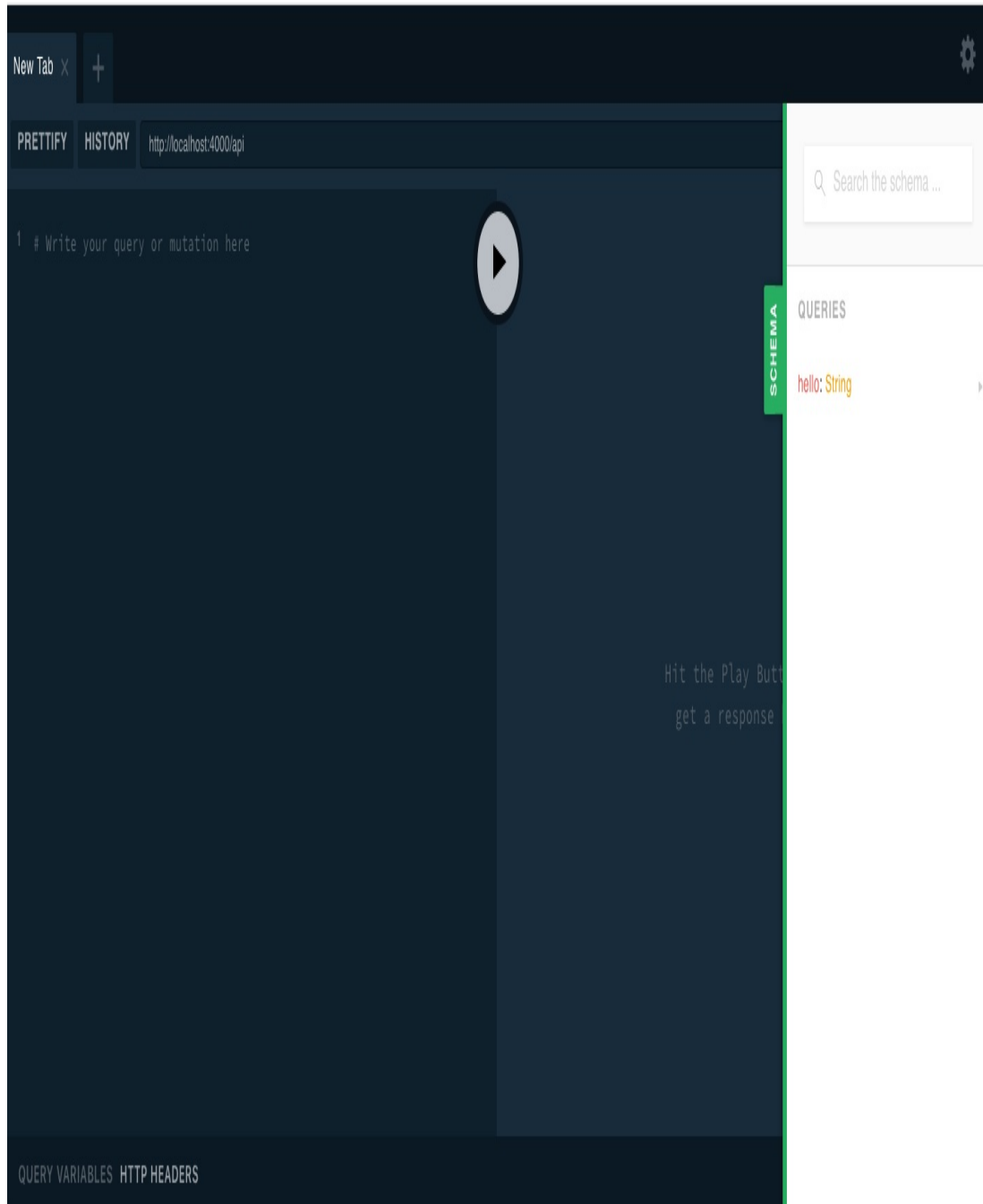


Figure 4-1. The GraphQL Playground

We can now write our query against our GraphQL API. To do so, we'll type the following into the the GraphQL Playground:

```
query {  
  hello  
}
```



When we click the Play button, our query should return the following:

```
{
  "data": {
    "hello": "Hello world!"
  }
}
```

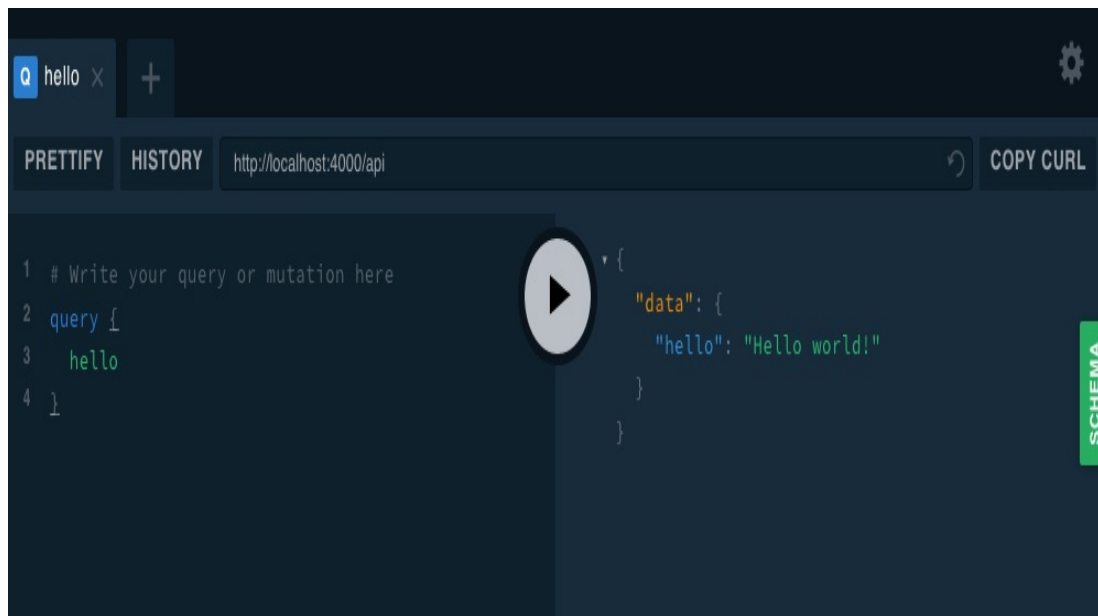


Figure 4-2. The hello query

And that's it! We now have a working GraphQL API that we've accessed via the GraphQL Playground. Our API takes a query of `hello` and returns the string `Hello world!`. More importantly, we now have the structure in place to build a fully featured API.

GraphQL Basics

In the previous section we dove right in and developed our first API, but let's take a few moments to step back and look at the different pieces of a GraphQL API. The two primary building blocks of a GraphQL API are

schemas and **resolvers**. By understanding these two components, we can apply them to our API design and development.

Schemas

A schema is a written representation of our data and interactions. By requiring a schema, GraphQL enforces us to follow a strict plan for our API. This is because our API can only return data and perform interactions that are defined within our schema.

The fundamental component of GraphQL schemas are object types. In our previous example we created a GraphQL object type of `Query` with a field of `hello`, which returned a scalar type of `String`. GraphQL contains five built-in scalar types:

- `String`: A string with UTF-8 character encoding
- `Boolean`: A true or false value
- `Int`: A 32-bit integer
- `Float`: A floating point value
- `ID`: A unique identifier

With these basic components we can construct a schema for an API. We do so by first defining the type. Let's imagine that we were creating an API for a pizza menu. In doing so, we might define a GraphQL schema type of `Pizza` like so:

```
type Pizza {  
}
```



Now, each pizza has a unique ID, a size (such as small, medium, and

large), a number of slices, and optional toppings. To write this, our Pizza schema might look something like this:

```
type Pizza {  
  id: ID  
  size: String  
  slices: Int  
  toppings: [String]  
}
```

In our schema, some field values are required (such as ID, size, and slices), while others may be optional (such as toppings). We can express that a field must contain a value by using an exclamation mark. Let's update our schema to represent the required values:

```
type Pizza {  
  id: ID!  
  size: String!  
  slices: Int!  
  toppings: [String]  
}
```

In this book, we'll be writing a basic schema, which will enable us to perform the vast majority of operations found in a common API. If you'd like to explore all of the GraphQL schema options, I'd encourage you to read the [GraphQL schema documentation](#).

Resolvers

The second piece of our GraphQL API will be resolvers. Resolvers perform exactly the action that they are advertised to do; they *resolve* the data that the API user has requested. We will write these resolvers by first defining them in our schema and then implementing the logic within our JavaScript code. Our API will contain two types of resolvers: queries and

mutations.

QUERIES

A query requests specific data from our API, in its desired format. In our hypothetical pizza API we may write a query that will return a full list of pizzas on the menu and another that might return detailed information about a single pizza. The query will then return an object, containing the data that the API user has requested. A query never modifies our data, only accesses it.

MUTATIONS

A mutation is used when we want to modify the data in our API. In our pizza example, we may write a mutation that changes the toppings for a given pizza and another that allows us to adjust the number of slices. Similar to a query, a is also expected to return a result in the form of an object, typically the end result of the performed action.

Adapting our API

Now that we have a good understanding of the components of GraphQL, let's adapt our initial API code for our notes application. To begin, we'll write some code to read and create notes.

The first thing we'll need is a little bit of data for our API to work with. Let's create an array of "note" objects, that we'll use as the basic data served by our API. As our project evolves, we'll replace this in memory data representation with a database. For now, we will store our data in a variable named `notes`. Each note in the array will be an object with three properties, `id`, `content`, and `author`:

```
let notes = [  
  {  
    id: '1',  
    content: 'This is a note',  
    author: 'Adam Scott'  
  },  
  {  
    id: '2',  
    content: 'This is another note',  
    author: 'Harlow Everly'  
  },  
  {  
    id: '3',  
    content: 'Oh hey look, another note!',  
    author: 'Riley Harrison'  
  }  
];
```

Now that we have some data, we will adapt our GraphQL API to work with that data. Let's begin by focusing on our schema. Our schema is GraphQL's representation of our data and how it will be interacted with. We know that we will have notes, which will be queried and mutated. These notes will, for now, contain an ID, content, and an author field. Let's create a corresponding note type within our `typeDefs` GraphQL schema. This will represent the properties of a note within our API:

```
type Note {  
  id: ID!  
  content: String!  
  author: String!  
}
```

Now, let's add a query that will allow us to retrieve the list of all notes. Let's update the `Query` type, to include a `notes` query, which will return the array of note objects:


```
type Query {  
  hello: String!  
  notes: [Note!]!  
}
```

Now, we can update our resolver code to perform the work of returning the array of data. Let's update our Query code to include the following `notes` resolver, which returns our raw data object:

```
Query: {  
  hello: () => 'Hello world!',  
  notes: () => notes  
},
```

If we now go to the GraphQL playground, running at <http://localhost:4000/api>, we can test our `notes` query. To do so, we can type the following query:

```
query {  
  notes {  
    id  
    content  
    author  
  }  
}
```

If we then click the **Play** button, we should see a `data` object returned, which contains our data array.

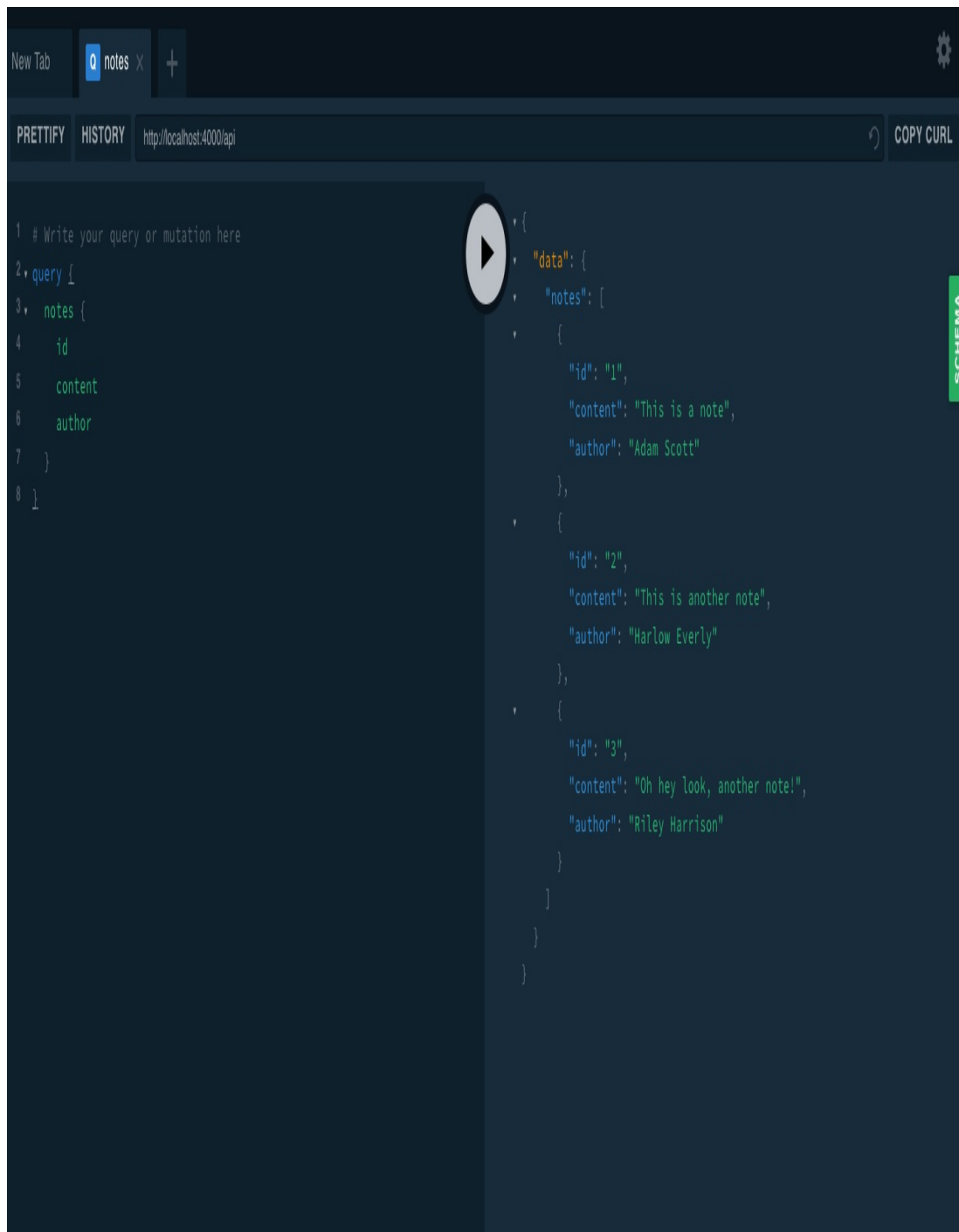


Figure 4-3. The notes query

Now, to try out one of the coolest aspects of GraphQL, we can remove any of our requested fields, such as `id` or `author`. When we do so, we see that our API returns precisely the data that we've requested. This allows the client that consumes the data to control the amount of data sent within

each request and limit that data to exactly what is required.

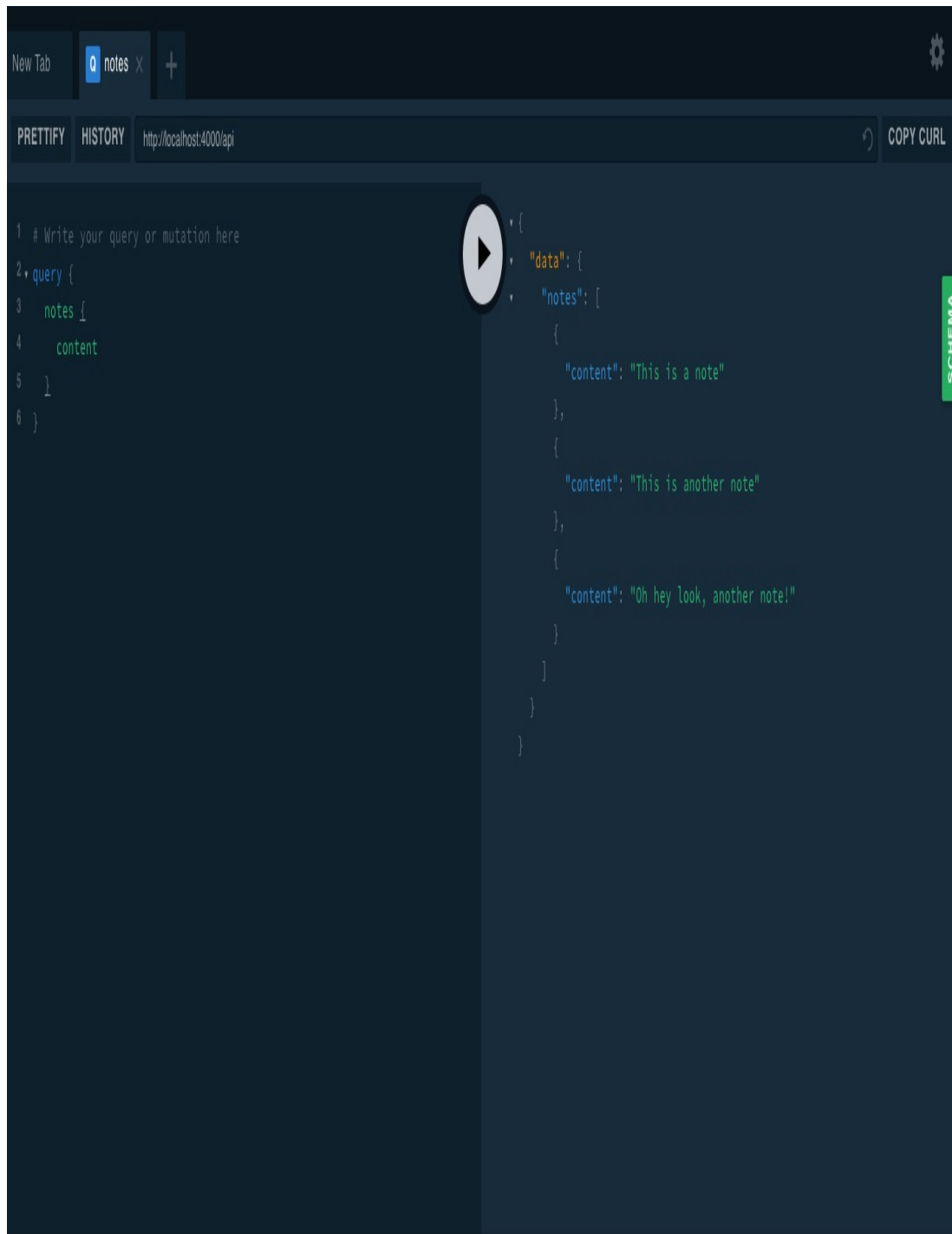


Figure 4-4. A notes query with only content data requested

Now that we can query our full list of notes, let's write some code that will allow us to query for a single note. We can imagine the usefulness of this

from a user interface perspective, for displaying a view that contains a single, specific note. To do so, we'll want to request a note with a specific `id` value. This will require us to use an **argument** in our GraphQL schema. An argument allows the API consumer to pass specific values to our resolver function, providing the necessary information for it to resolve. Let's go ahead and add a `note` query, which will take an argument of `id`, with the type `ID`. We'll update our `Query` object within our `typeDefs` to the following, which includes our new `note` query:

```
type Query {  
  hello: String  
  notes: [Note!]!  
  note(id: ID): Note!  
}
```

Now that we've updated our schema, we are able to write a query resolver to return the requested note. To do this, we'll need to be able to read the API user's argument values. Helpfully, Apollo Server passes useful parameters to our resolver functions. These are as follows:

- **parent:** The result of the parent query, which is useful when nesting queries.
- **args:** These are the arguments passed by the user in the query.
- **context:** Information passed along from the server application to the resolver functions. This could include things such as the current user or database information.
- **info:** Information about the query itself.

We'll be exploring these further as needed within our code. If you're curious, you can learn more about these parameters in [Apollo Server's documentation](#). For now, we'll only be needing the information contained

within the second parameter, `args`.

Our `note` query will take the `note id` as an argument and find it within our array of `note` objects. To do so, let's add the following to our query resolver code:

```
note: (parent, args) => {  
  return notes.find(note => note.id === args.id);  
}
```

Our resolver code should now look as follows:

```
const resolvers = {  
  Query: {  
    hello: () => 'Hello world!',  
    notes: () => notes,  
    note: (parent, args) => {  
      return notes.find(note => note.id === args.id);  
    }  
  }  
};
```

To run our query, let's go back to our web browser and visit the GraphQL Playground at <http://localhost:4000/api>. We can now query for a note with a specific `id` as follows:

```
query {  
  note(id: "1") {  
    id  
    content  
    author  
  }  
}
```

When we run this query, we should receive the results of a note with the

requested `id` value. If we attempt to query for a note that doesn't exist, we should receive a result with a value of `null`. To test this, try changing the `id` value to return different results.

We can now query our list of notes as well as an individual note. Let's wrap up our initial API code by introducing the ability to create a new note, using a GraphQL mutation. In that mutation, the user will pass in the note's content. For now, we'll hard code the author of the note. Let's begin by updating our `typeDefs` schema with a `Mutation` type, which we will call `newNote`:

```
type Mutation {  
  newNote(content: String!): Note!  
}
```

We'll now write a mutation resolver, which will take in the note content as an argument, store the note as an object, and add it in memory to our `notes` array. To do this, we'll add a `Mutation` object to our `resolvers`. Within the mutation object, we'll add a function called `newNote`, with `parent` and `args` parameters. Within this function, we'll take the argument content and create an object with `id`, `content`, and `author` keys. As you may have noticed, this matches the current schema of a note. We will then push this object to our `notes` array and return the object. Returning the object allows the GraphQL mutation to receive a response in the intended format. Let's go ahead and write this code as follows:

```
Mutation: {  
  newNote: (parent, args) => {  
    let noteValue = {  
      id: notes.length + 1  
      content: args.content,  
      author: 'Adam Scott'  
    }
```

```
};  
notes.push(noteValue);  
return noteValue;  
}  
}
```

Our `src/index.js` will now read as follows:

```
const express = require('express');  
const { ApolloServer, gql } = require('apollo-server-express');  
  
// Run our server on a port specified in our .env file or port 4000  
const port = process.env.PORT || 4000;  
  
let notes = [  
  {  
    id: '1',  
    content: 'This is a note',  
    author: 'Adam Scott'  
  },  
  {  
    id: '2',  
    content: 'This is another note',  
    author: 'Harlow Everly'  
  },  
  {  
    id: '3',  
    content: 'Oh hey look, another note!',  
    author: 'Riley Harrison'  
  }  
];  
  
// Construct a schema, using GraphQL's schema language  
const typeDefs = gql`  
  type Note {  
    id: ID!  
    content: String!  
    author: String!  
  }  
  
  type Query {  
    hello: String  
  }  
`;
```

```

    notes: [Note!]!
    note(id: ID!): Note!
  }

  type Mutation {
    newNote(content: String!): Note!
  }
`
;

// Provide resolver functions for our schema fields
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  },
  Mutation: {
    newNote: (parent, args) => {
      let noteValue = {
        id: notes.length + 1,
        content: args.content,
        author: 'Adam Scott'
      };
      notes.push(noteValue);
      return noteValue;
    }
  }
};

const app = express();

// Apollo Server setup
const server = new ApolloServer({ typeDefs, resolvers });

// Apply the Apollo GraphQL middleware and set the path to /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);

```



```
);
```

With our schema and resolver updated to accept a mutation, let's try it out in GraphQL Playground at <http://localhost:4000/api>. In the playground, click the + sign to create a new tab and write the mutation as follows:

```
mutation {  
  newNote (content: "This is a mutant note!") {  
    content  
    id  
    author  
  }  
}
```

When we click the play button, we should receive a response containing the content, id, and author of our new note. We can also see that our mutation worked by re-running our `notes` query. To do so, either switch back to the GraphQL Playground tab containing that query, or type the following:

```
query {  
  notes {  
    content  
    id  
    author  
  }  
}
```

When this query runs, we should now see four notes, including our recently added one.

NOTE

We are currently storing our data in memory. This means that anytime we restart our server, we will lose that data. We'll be persisting our data using a database in the next

chapter.

We've now successfully implemented our query and mutation resolvers and tested them within the GraphQL Playground user interface.

Conclusion

In this chapter we've successfully built a GraphQL API, using the `apollo-server-express` module. We are now able to run queries and mutations against an in-memory data object. This set up will provide us a solid foundation on which to build any API. In the next chapter we'll explore the ability to persist data by using a database.