Andrew Mead

Advanced Node.js Development

Master Node.js by building real-world applications

Packt>

Advanced Node.js Development

Master Node.js by building real-world applications

Andrew Mead

BIRMINGHAM - MUMBAI

Advanced Node.js Development

Acquisition Editor: Ben Renow-Clarke
Content Development Editor: Monika Sangwan
Technical Editor: Gaurav Gavas
Copy Editor: Tom Jacob
Project Coordinator: Suzanne Coutinho
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Coordinator: Shantanu N. Zagade

First published: March 2018

Production reference: 1290318

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78839-393-5

www.packtpub.com

Packt Upsell

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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

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

Contributors

About the author

Andrew Mead is a full-stack developer living in beautiful Philadelphia! He launched his first Udemy course in 2014 and had a blast teaching and helping others. Since then, he has launched three courses with over 21,000 students and over 1,900 5-star reviews.

Andrew currently teaches Node.js, Gulp, and React. Before he started teaching, he created a web app development company. He has helped companies of all sizes launch production web applications to their customers. He has had the honor of working with awesome companies such as Siemens, Mixergy, and Parkloco. He has a Computer Science degree from Temple University, and he has been programming for just over a decade. He loves creating, programming, launching, learning, teaching, and biking.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Preface

Welcome to *Advanced Node.js Development*. This book is packed with a ton of content, projects, challenges, and real-world examples, all designed to teach you Node by *doing*. This means you'll be getting your hands dirty early on in the upcoming chapters writing some code, and you'll be writing code for every project. You will be writing every line of code that powers our applications. Now, we would require a text editor for this book.

All the projects in the book are fun to build and they were designed to teach you everything required to launch your own Node app, from planning to development and testing to deploying. Now, as you launch these different Node applications and move through the book, you will run into errors, which is bound to happen. Maybe something doesn't get installed as expected, or maybe you try to run an app and instead of getting the expected output, you get a really long obscure error message. Don't worry, I am there to help. I'll show you tips and tricks to get pass through those errors in the chapters. Let's go ahead and get to it.

Who this book is for

This book targets anyone looking to launch their own Node applications, switch careers, or freelance as a Node developer. You should have a basic understanding of JavaScript in order to follow this book.

What this book covers

`Chapter 1`, *Getting Set Up*, will be a very basic setup for your local environments. We'll learn to install MongoDB and Robomongo.

`Chapter 2`, *MongoDB, Mongoose, and REST APIs – Part 1*, will help you learn how to connect your Node applications to the MongoDB database you've been running on your local machine.

`Chapter 3`, *MongoDB, Mongoose, and REST APIs – Part 2*, will help you start playing with Mongoose and connect to our MongoDB database.

`Chapter 4`, *MongoDB, Mongoose, and REST APIs – Part 3*, will resolve queries and ID validation after playing with Mongoose.

`Chapter 5`, *Real-Time Web Apps with Socket.io*, will help you learn in detail about Socket.io and WebSockets, help you and create real-time web applications.

`Chapter 6`, *Generating newMessage and newLocationMessage*, discusses how to generate text and gelocation messages.

`Chapter 7`, *Styling Our Chat Page as a Web App*, continues our discussion on styling our chat page and make it look more like a real web application.

`Chapter 8`, *The Join Page and Passing Room Data*, continues our discussion about the chat page and look into the join page and passing room data.

`Chapter 9`, *ES7 classes*, will help you learn the ES6 class syntax and using it creating user's class and some other methods.

`Chapter 10`, *Async/Await Project Setup*, will walk you through the process of learning how async/await works.

To get the most out of this book

To run the projects in this book, you will need the following:

- The latest version of Node.js (9.x.x at the time of writing this book)
- Express
- MongoDB
- Mongoose
- Atom

We'll see the rest of the requirements along the course of the book.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Advanced-Node.js-Development`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/AdvancedNode.jsDevelopment_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {
 height: 100%;
 margin: 0;
 padding: 0
}
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Getting Set Up

In this chapter, you'll get your local environment set up for the rest of the book. Whether you're on macOS, Linux, or Windows, we'll install MongoDB and Robomongo.

More specifically, we'll cover the following topics:

- MongoDB and Robomongo installation for Linux and macOS
- MongoDB and Robomongo installation for Windows

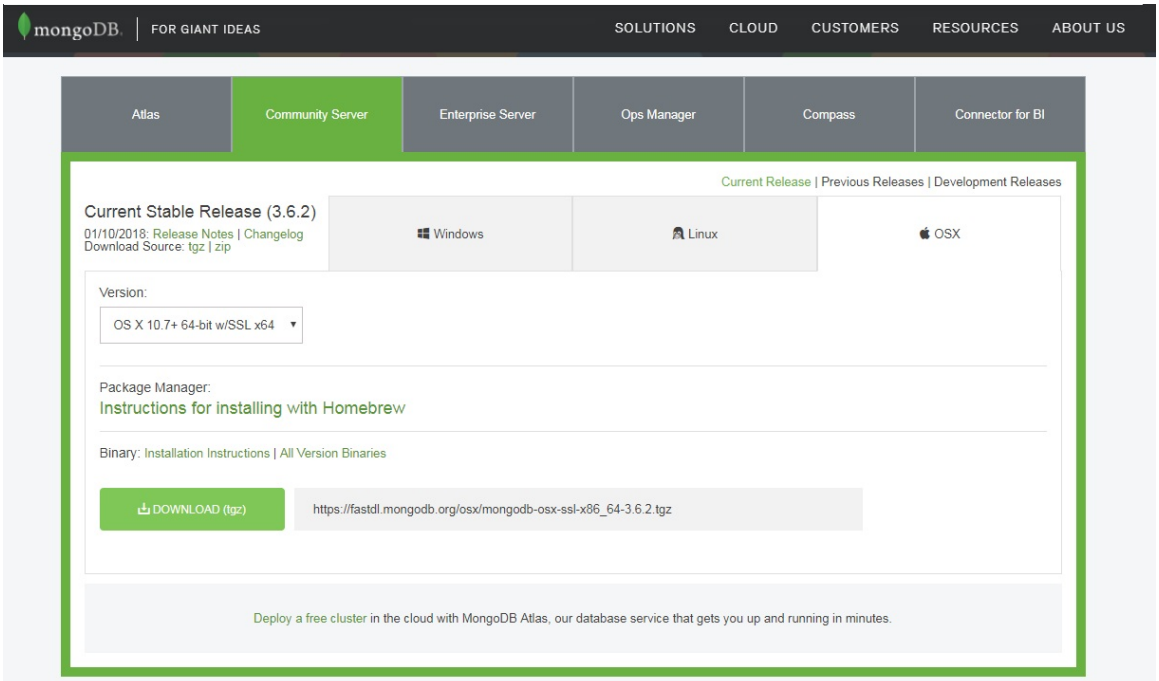
Installing MongoDB and Robomongo for Linux and macOS

This section is for macOS and Linux users. If you are on Windows, I have written a separate section for you.

The first thing we'll do is to download and set up MongoDB, as this will be the database we will use. We'll be using a third-party service to host our database when we eventually deploy it to Heroku, but on our local machine we'll need to download MongoDB so that we can start up a database server. This will let us connect to it via our Node applications to read and write data.

In order to grab the database, we'll head over to `mongodb.com`. Then we can go to the Download page and download the appropriate version.

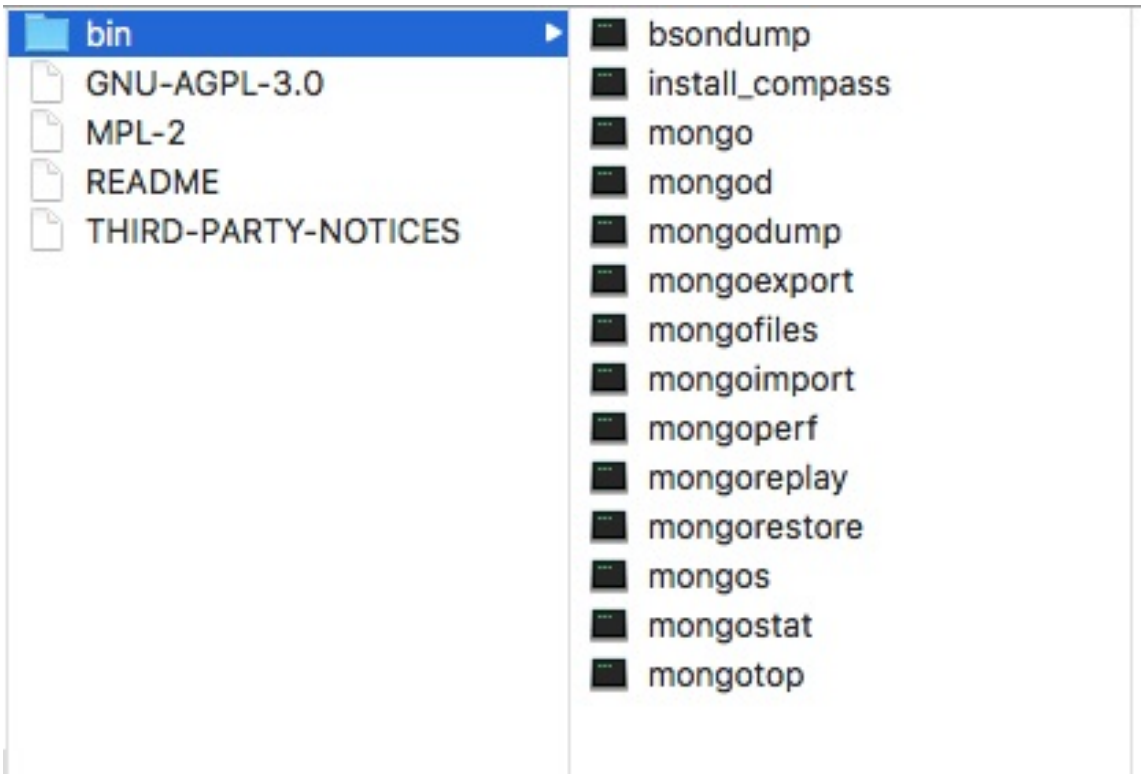
On this page, scroll down and select Community Server; this is the one we'll be using. Also, there are options for different operating systems, whether it's Windows, Linux, macOS, or Solaris. I'm on macOS, so I'll use this download:



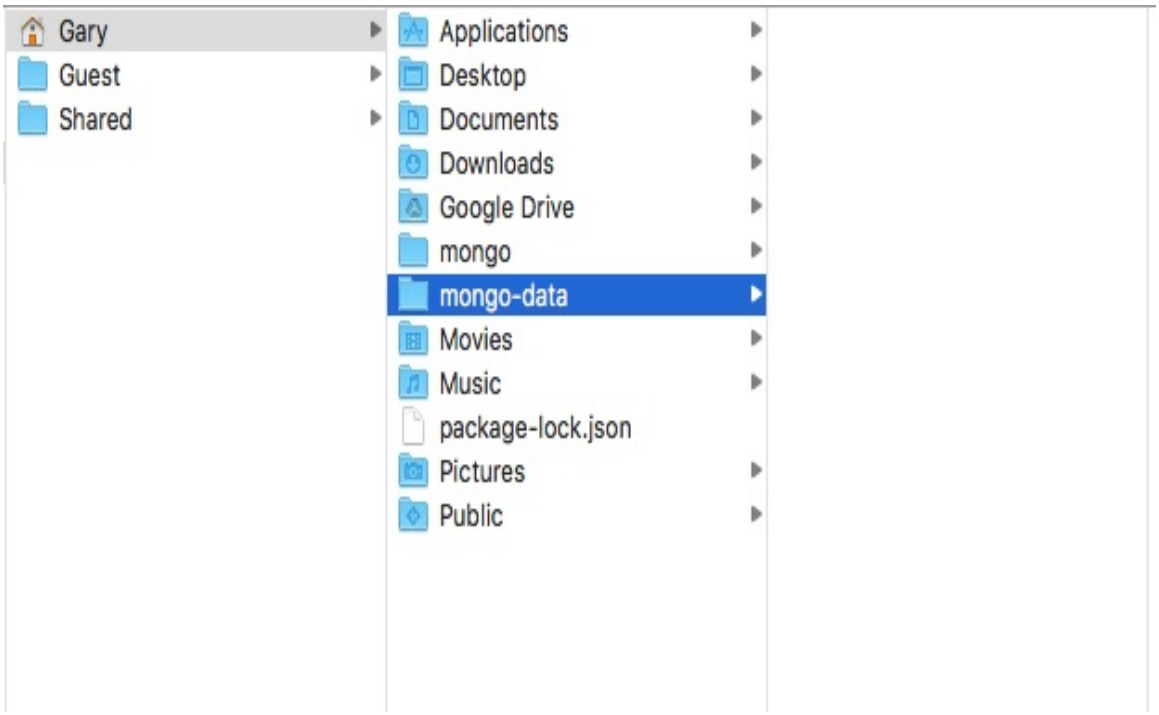
If you're on Linux, click on Linux; then go to the Version drop down and select the appropriate version. For example, if you're on Ubuntu 14.04, you can download the correct one from the Linux tab. Then, you can simply click on the Download button and follow along.

Next you can open it up. We'll just extract the directory, creating a brand new folder in the `Downloads` folder. If you're on Linux, you might need to manually extract the contents of that archive into the `Downloads` folder.

Now this folder contains a `bin` folder, and in there we have all of the executables that we need in order to do things such as connecting to the database and starting a database server:



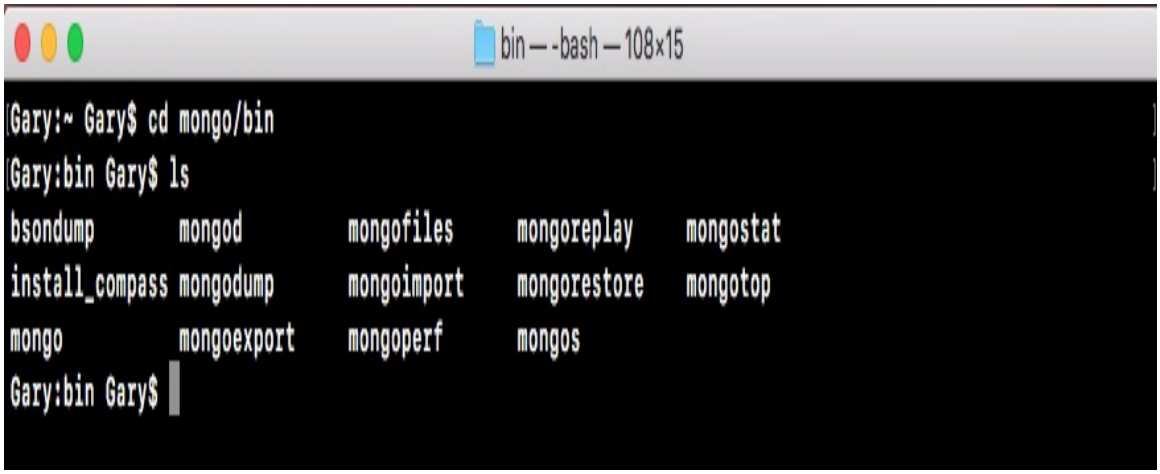
Before we go ahead and run any of them. We'll rename this directory to `mongo` and then move it into the `user` directory. You can see that now in the `user` directory, I have the `mongo` folder. We'll also create a brand new directory alongside of `mongo` called `mongo-data`, and this will store the actual data inside of the database:



So when we insert a new record into the `Todos` table, for example, that will live in the `mongo-data` folder. Once you have the `mongo` folder moved into the `user` directory and you have the new `mongo-data` folder, you are ready to actually run the database server from Terminal. I'll go into Terminal and navigate into that brand new `mongo` folder that is in the `user` directory, where I currently am, so I can `cd` into `mongo`, then I'll `cd` into the `bin` directory by tacking it on right there:

```
cd mongo/bin
```

From here, we have a bunch of executables that we can run:

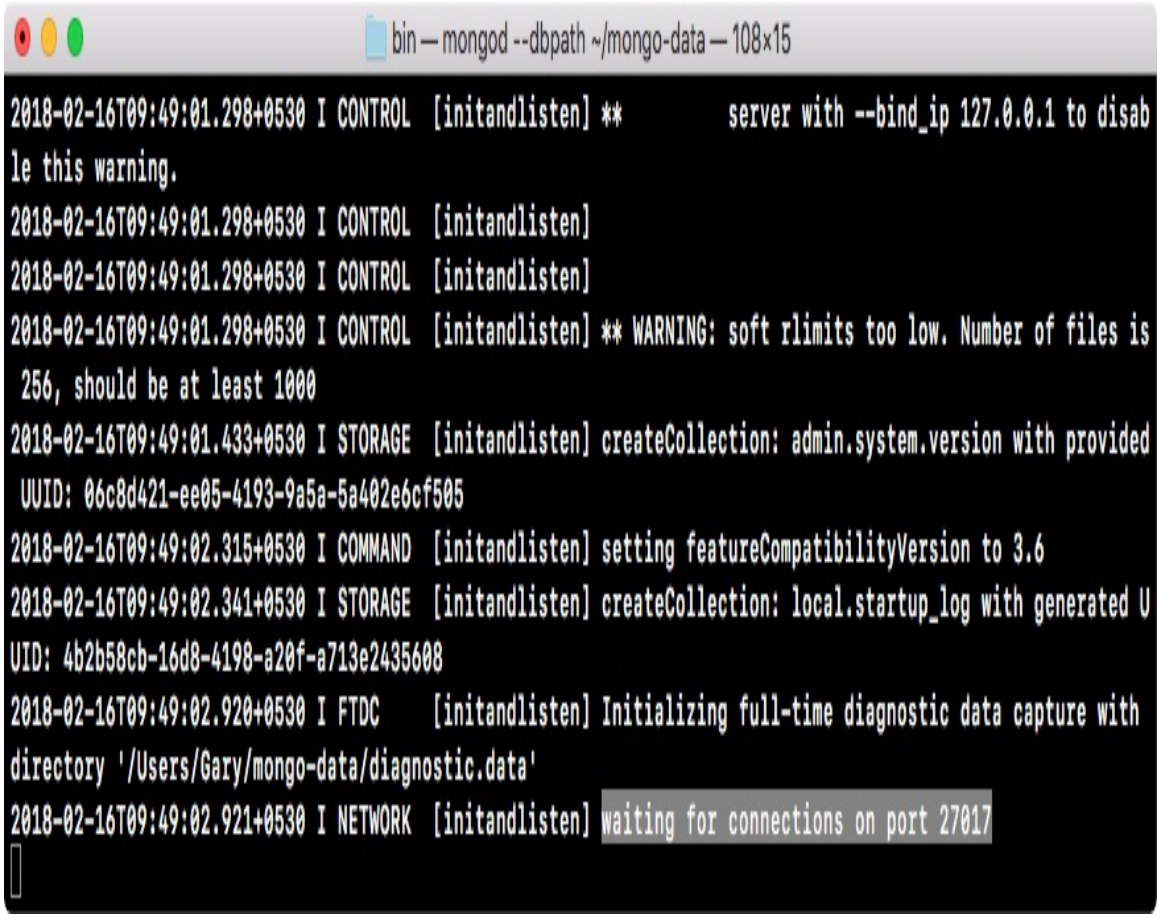


We have things such as bisondump and mongodump. In this section, we'll focus on: mongod, which will start up the database server, and mongo, which will let us connect to the server and run some commands. Just like when we type `node` we can run some JavaScript commands right in Terminal, when we type `mongo`, we'll be able to run some Mongo commands to insert, fetch, or do anything we like with the data.

First up though, let's start up the database server. I'll use `./` to run a file in the current directory. The file we'll run is called `mongod`; also, we do need to provide one argument: the `dbpath` argument. The `dbpath` argument will get set equal to the path of the directory we just created, the `mongo-data` directory. I'll use `~` (the tilde) to navigate to the user directory, and then to `/mongo-data`, as shown here:

```
./mongod --dbpath ~/mongo-data
```

Running this command will start up the server. This will create an active connection, which we can connect to for manipulating our data. The last line that you see when you run the command should be, waiting for connections on port 27017:



If you see this, it means that your server is up and running.

Next up, let's open a new tab, which starts in the exact same directory, and this time around, instead of running `mongod`, we'll run the `mongo` file:

```
./mongo
```

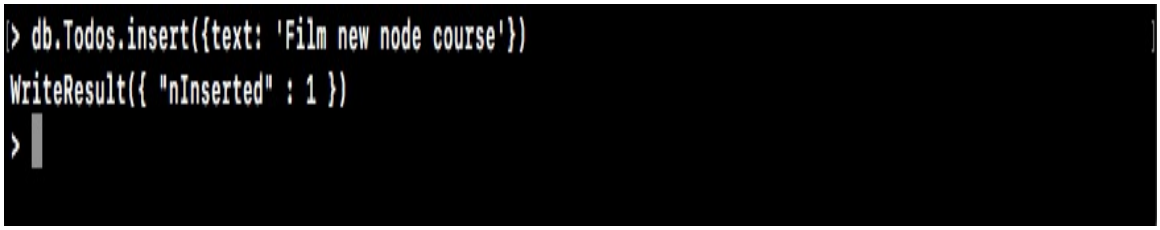
When we run `mongo`, we open up a console. It connects to the database server we just started, and from here, we can start running some commands. These commands are just to test that things are working as expected. We'll be going over all of this in detail later in this section. For now though, we can access `db.Todos`, and then we'll call `.insert` to create a brand new Todo record. I'll call it like a function:

```
db.Todos.insert({})
```

Next, inside of `insert`, we'll pass in our document. This will be the MongoDB document we want to create. For now, we'll keep things really simple. On our object, we'll specify one attribute, `text`, setting it equal to a string. Inside of quotes, type anything you want to do. I'll say `Film new node course`:

```
db.Todos.insert({text: 'Film new node course'})
```

With your command looking just like this, you can press *enter*, and you should get back a WriteResult object with an nInserted property, which is short for the number inserted: a value set to 1. This means that one new record was created, and that is fantastic!

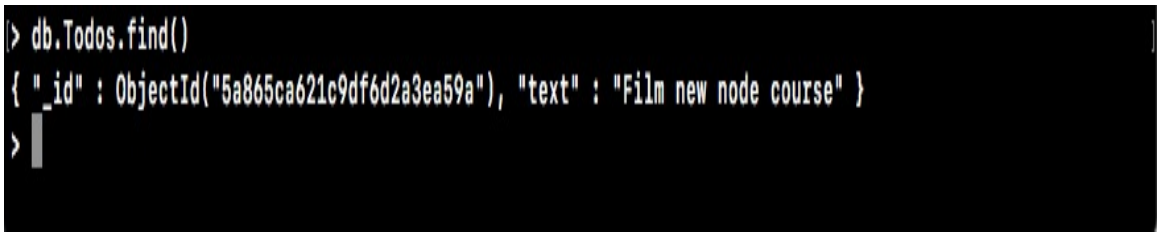


Now that we've inserted a record, let's fetch the record just to make sure that everything worked as expected.

Instead of calling `insert`, we'll call `find` without any arguments. We want to return every single item in the `Todos` collection:

```
db.Todos.find()
```

When I run this, what do we get? We get one object-looking thing back:



We have our `text` attribute set to the text that we provided, and we have an `_id` property. This is the unique identifier for each record, which we'll talk about later. As long as you're seeing the text property coming back to what you set, you are good to go.

We can shut down the `mongo` command. However, we will still leave the `mongod` command running because there's one more thing I want to install. It's called Robomongo, and it's a graphic user interface for managing your Mongo database. This will be really useful as you start playing around with Mongo. You'll be able to view the exact data saved in the database; you can manipulate it and do all sorts of stuff.

Over in **Finder**, we have our `mongo-data` directory, and you can see that there is a ton of stuff in here. This means that our data was successfully saved. All of the data is in this `mongo-data` directory. To download and install Robomongo, which is available for Linux, Windows and macOS, we'll head over to `robomongo.org` and grab the installer for our operating system:

Secure | https://robomongo.org

Robo 3T

Download Blog Account

Studio 3T

Are you serious about MongoDB? Choose Studio 3T - our fully featured IDE for MongoDB professionals.

- Fully featured IDE with embedded shell
- Visual Query Builder
- In-Place editing
- IntelliShell with Auto-Completion
- Query MongoDB with SQL
- Export to / import from SQL DB
- Aggregation Pipeline Editor
- And so much more...

Download Studio 3T

Robo 3T

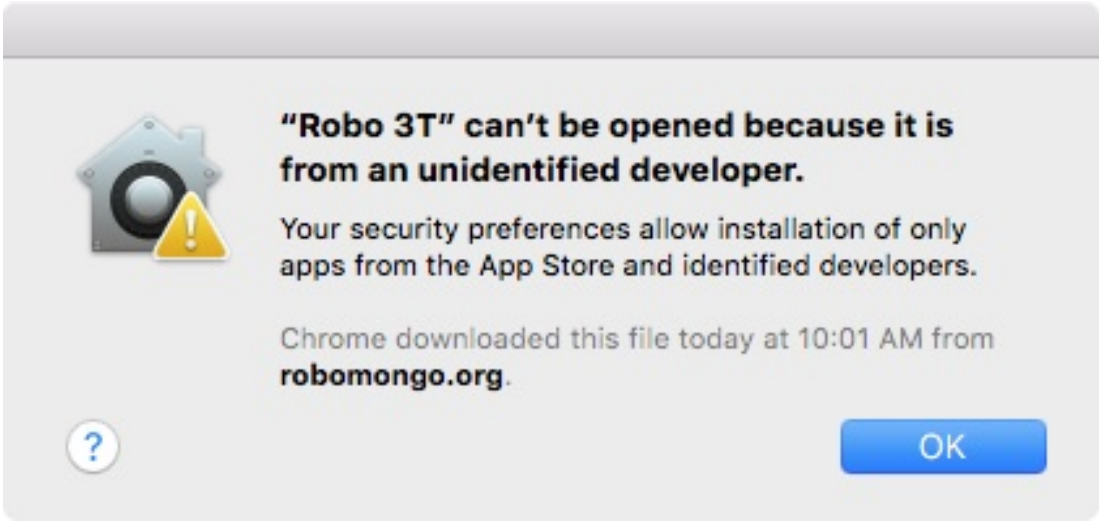
Robo 3T (formerly Robomongo) is the free lightweight GUI for MongoDB enthusiasts.

- MongoDB GUI with embedded shell

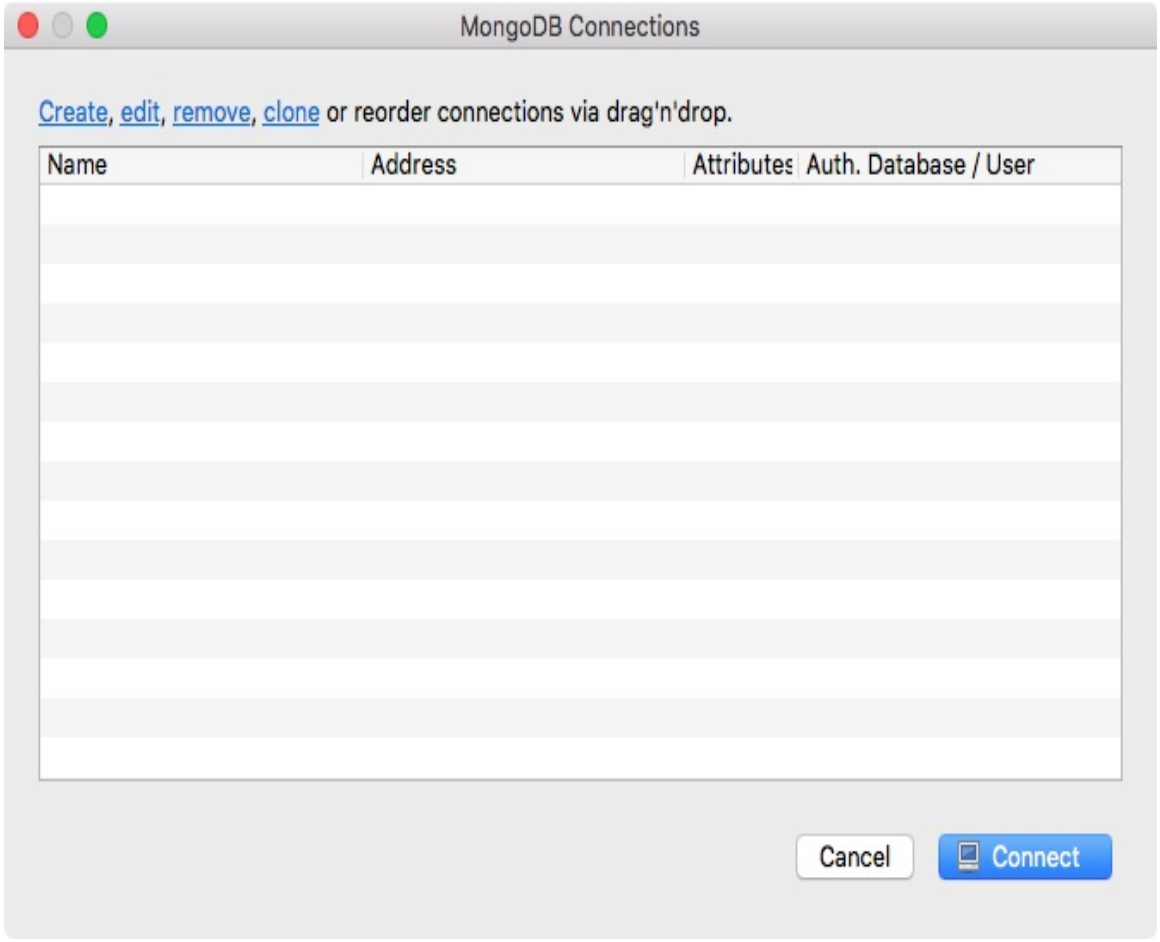
Download Robo 3T

We can click on Download Robo 3T and download the most recent version; it should automatically detect your OS. Download the installer for either Linux or macOS. The one for macOS is really simple. It's one of those installers where you take the icon and drag it into the `Applications` folder. For Linux, you'll need to extract the archive and run the program in the `bin` directory. This will start up Robomongo on your Linux distribution.

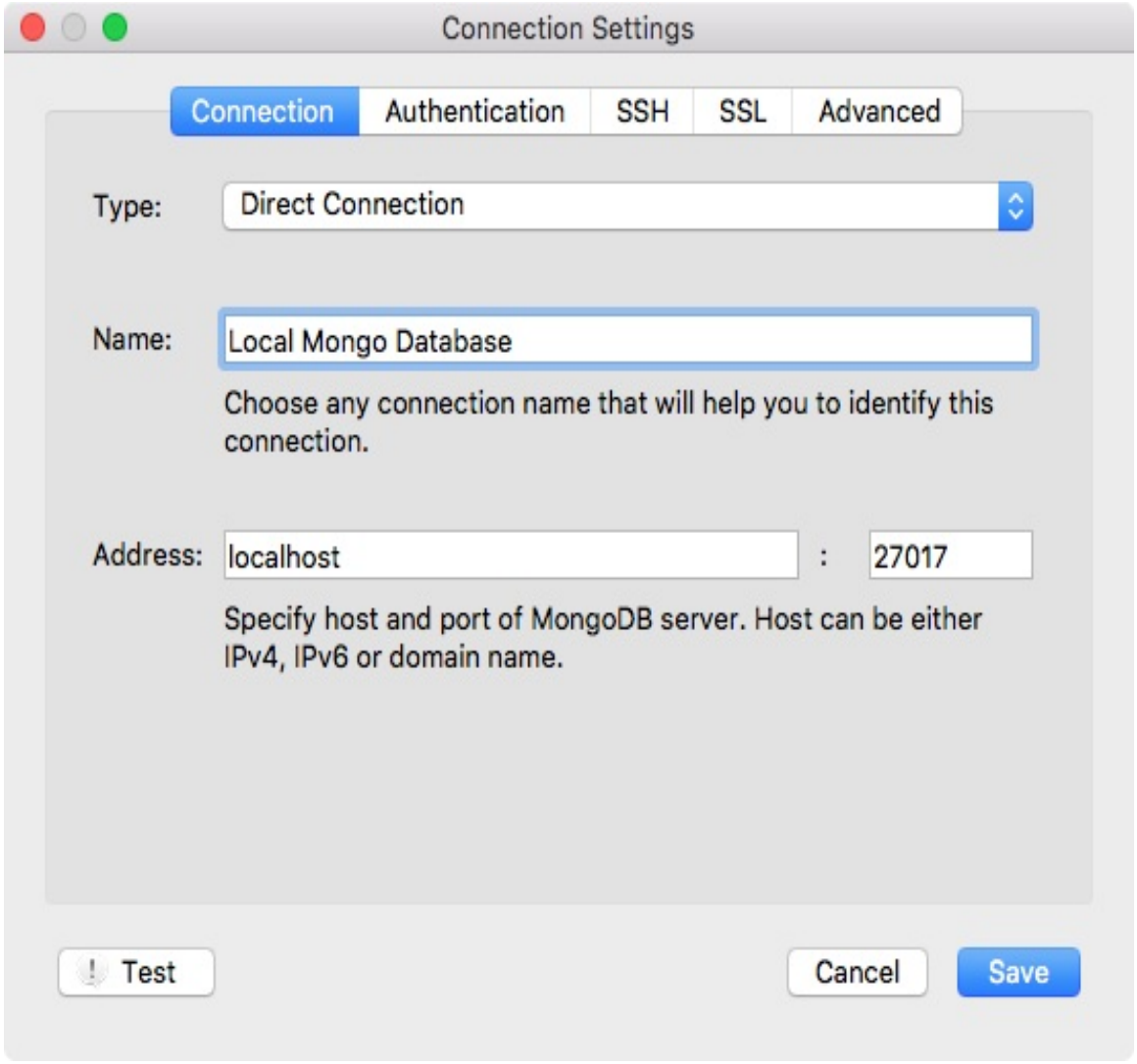
Since I'm using macOS, I'll just quickly drag the icon over to Applications, and then we can play around with the program itself. Next, I'll open it up inside the Finder. When you first open up Robomongo, you might get a warning like the following on macOS, since it's a program that we downloaded and it's not from an identified macOS developer:



This is fine; most programs you download from the web will not be official since they did not come from the App Store. You can right-click on the downloaded package, select Open, and then click on Open again to run that program. When you first open it, you'll see some screens like the following:

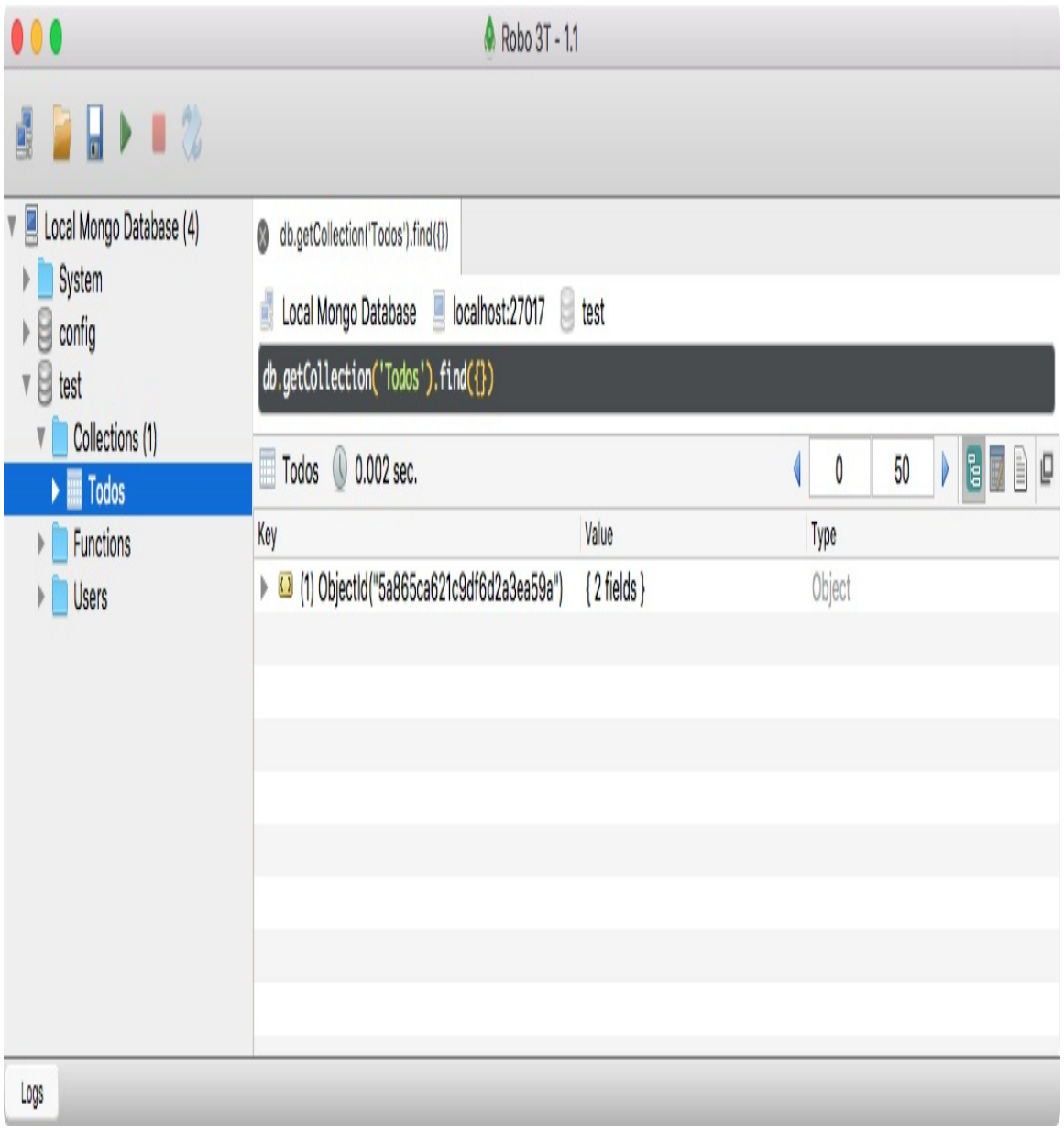


We have a little screen in the background and a list of connections; currently that list is empty. What we need to do is to create a connection for our local MongoDB database so that we can connect to it and manipulate that data. We have Create. I'll click on this, and the only thing we'll need to update is Name. I'll give it a more descriptive name, such as `Local Mongo Database`. I'll set Address to `localhost` and the `27017` port is correct; there's no need to change these. So, I'll click on Save:

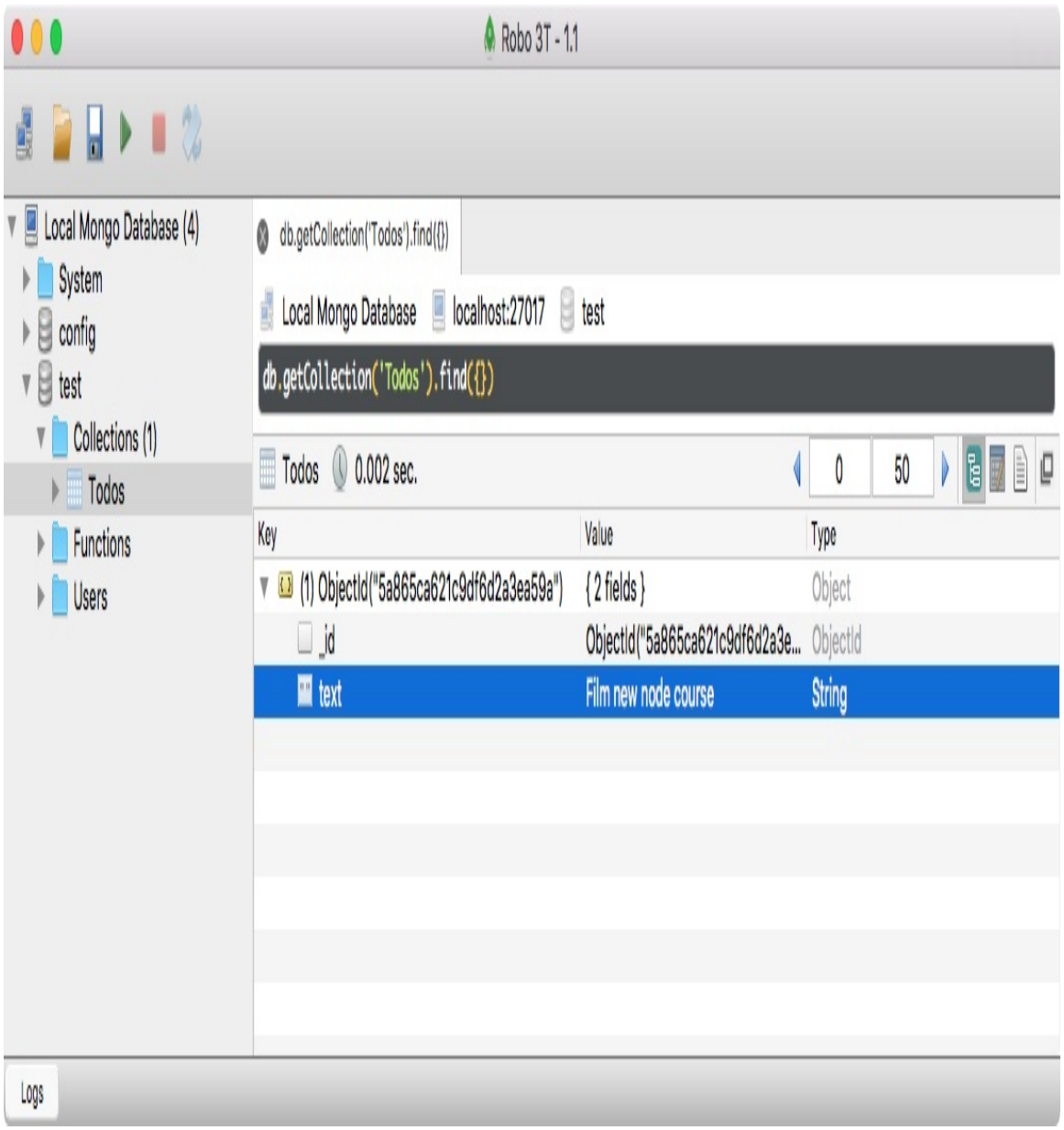


Next, I'll double-click on the database to connect to it. Inside the tiny window, we have our database. We are connected to it; we can do all sorts of things to manage it.

We can open up the `test` database, and in there, we should see one `Collections` folder. If we expand this folder, we have our `Todos` collection, and from there, we can right-click on the collection. Next, click on View Documents, and we should get our one Todo item, the one that we created over inside the Mongo console:



I can expand it to view the text property. Film new node course shows up:



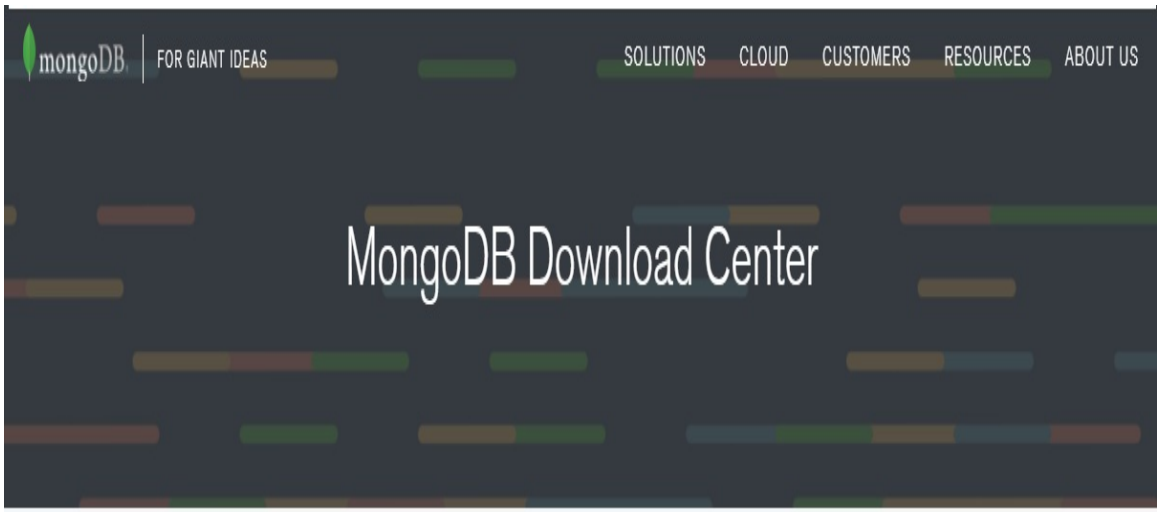
If you're seeing this, then you are done.

The next section is for Windows users.

Installing MongoDB and Robomongo for Windows

If you're on Windows, this is the installation section for you. If you're on Linux or macOS, the previous section was for you; you can skip this one. Our goal here is to install MongoDB on our machines, which will let us create a local MongoDB database server. We'll be able to connect to that server with Node.js, and we'll be able to read and write data to the database. This will be fantastic for the Todo API, which will be responsible for reading and writing various Todo-related information.

To get started, we'll grab the MongoDB installer by going over to `mongodb.com`. Here we can click on the big green Download button; also, we can see several options on this page:



Atlas | Community Server | Enterprise Server | Ops Manager | Compass | Connector for BI

Current Release | Previous Releases | Development Releases

Current Stable Release (3.6.2)

01/10/2018: Release Notes | Changelog
Download Source: tgz | zip

Windows | Linux | OSX

Version:

Windows Server 2008 R2 64-bit and later, with SSL support x64

Installation Package:

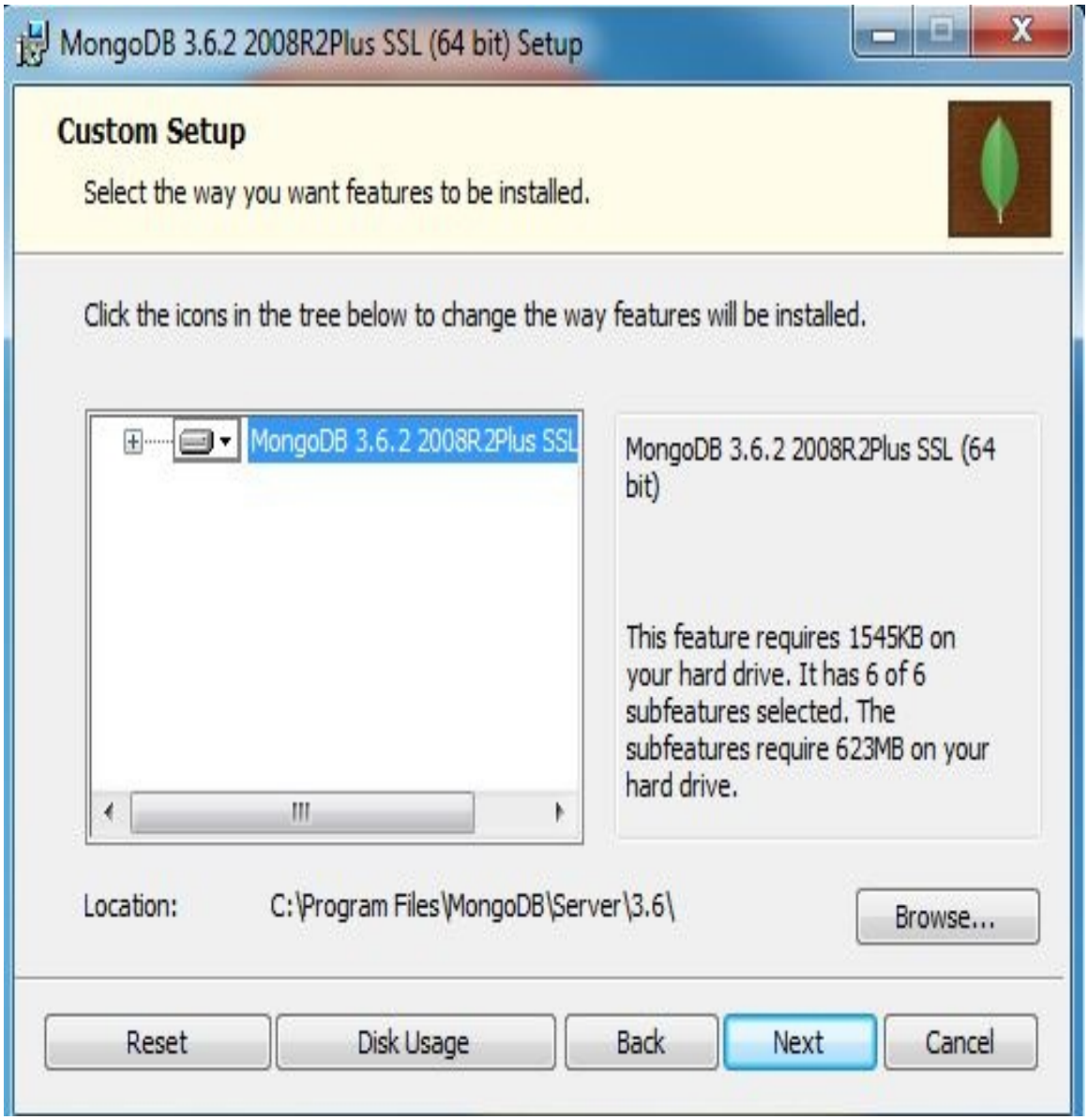
DOWNLOAD (msi)

Binary: Installation Instructions | All Version Binaries

Deploy a free cluster in the cloud with MongoDB Atlas, our database service that gets you up and running in minutes.

We'll use Community Server and for Windows. If you go to the Version drop down, none of the versions there will look right for you. The top one is what we want: Windows Server 08 R2 64-bit and later with SSL support. Let's start to download this. It is slightly big; just a tad over 100 MB, so it will take a moment for the download to begin.

I'll start it up. It's one of those basic installers, where you click on Next a few times and you agree to a license agreement. Click on the Custom option for a second, although we will be following through with the Complete option. When you click on Custom, it will show you where on your machine it's going to be installed, and this is important. Here, you can see that for me it's on `C:\Program Files\MongoDB\Server`, then in the `3.2` directory:



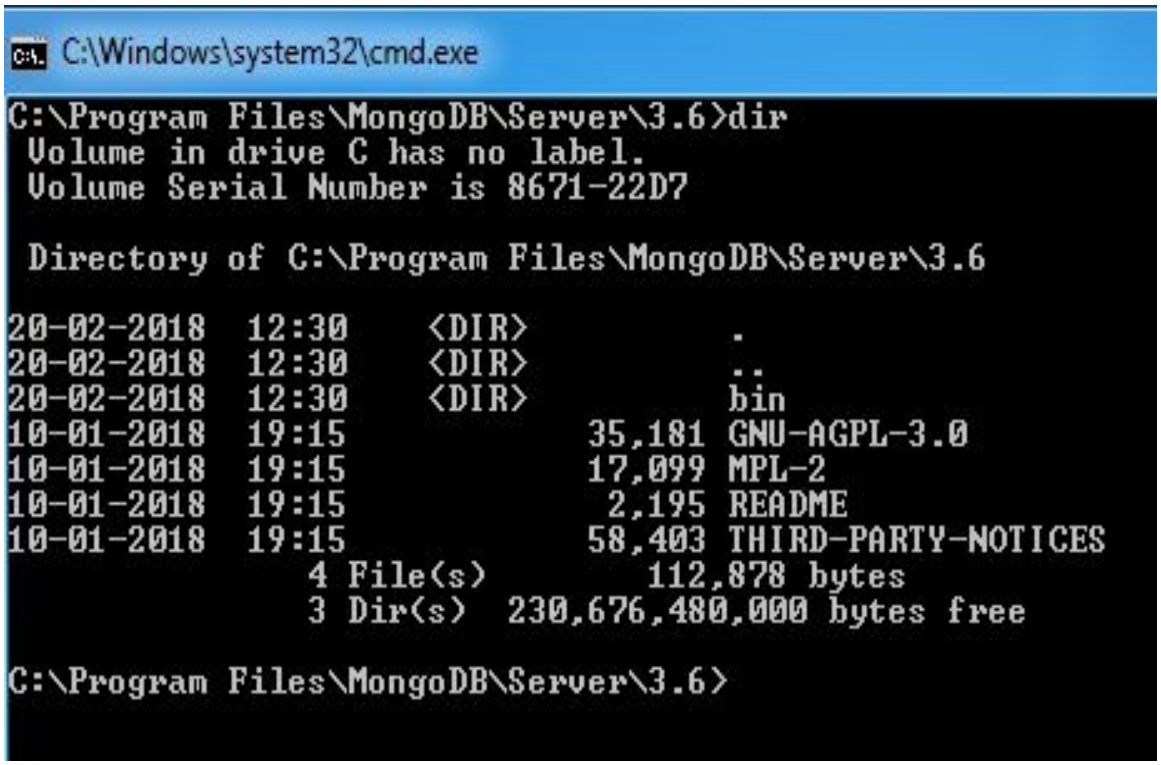
This is going to be important because we'll need to navigate into this directory in order to start up the MongoDB server. I will go back though, and I will be using the Complete option, which installs everything we need. Now we can actually start the installation process. Usually, you have to click on Yes, verifying that you want to install the software. I'll go ahead and do that, and then we are done.

Now once it's installed, we'll navigate into Command Prompt and boot up a server. The first thing we need to do is to navigate into that `Program Files` directory. I'm in Command Prompt. I recommend that you use Command Prompt and not Git Bash. Git Bash will not

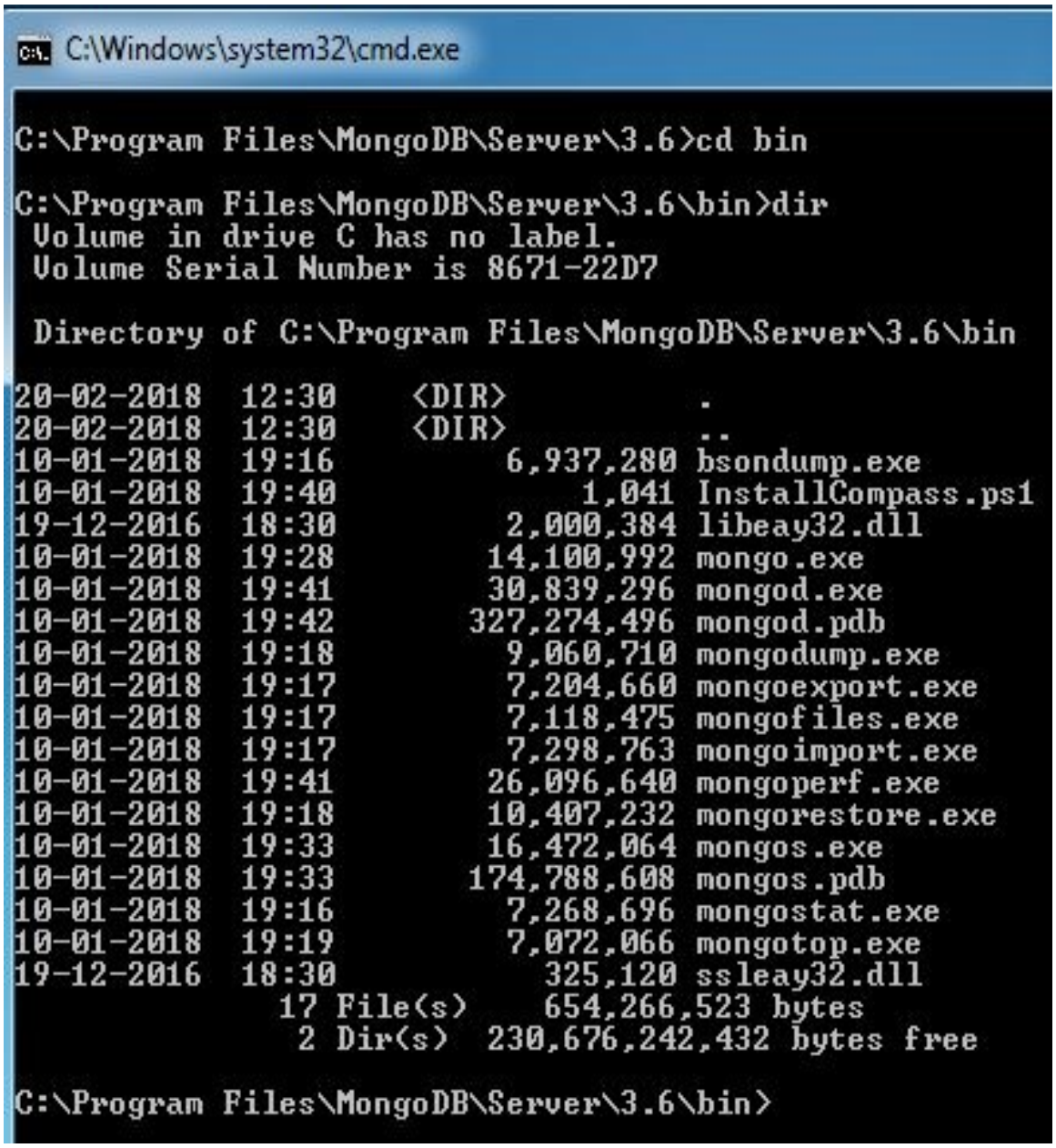
work for starting up the MongoDB server. I'll navigate to the root of my machine using `cd/`, and then we can start navigating to that path using the following command:

```
cd Program Files/MongoDB/Server/3.2
```

This is the directory where MongoDB was installed. I can use `dir` to print out the contents of this directory, and what we care about here is the `bin` directory:



We can navigate into `bin` using `cd bin`, and print its contents out using `dir`. Also, this directory contains a whole bunch of executables that we'll use to do things such as starting up our server and connecting to it:



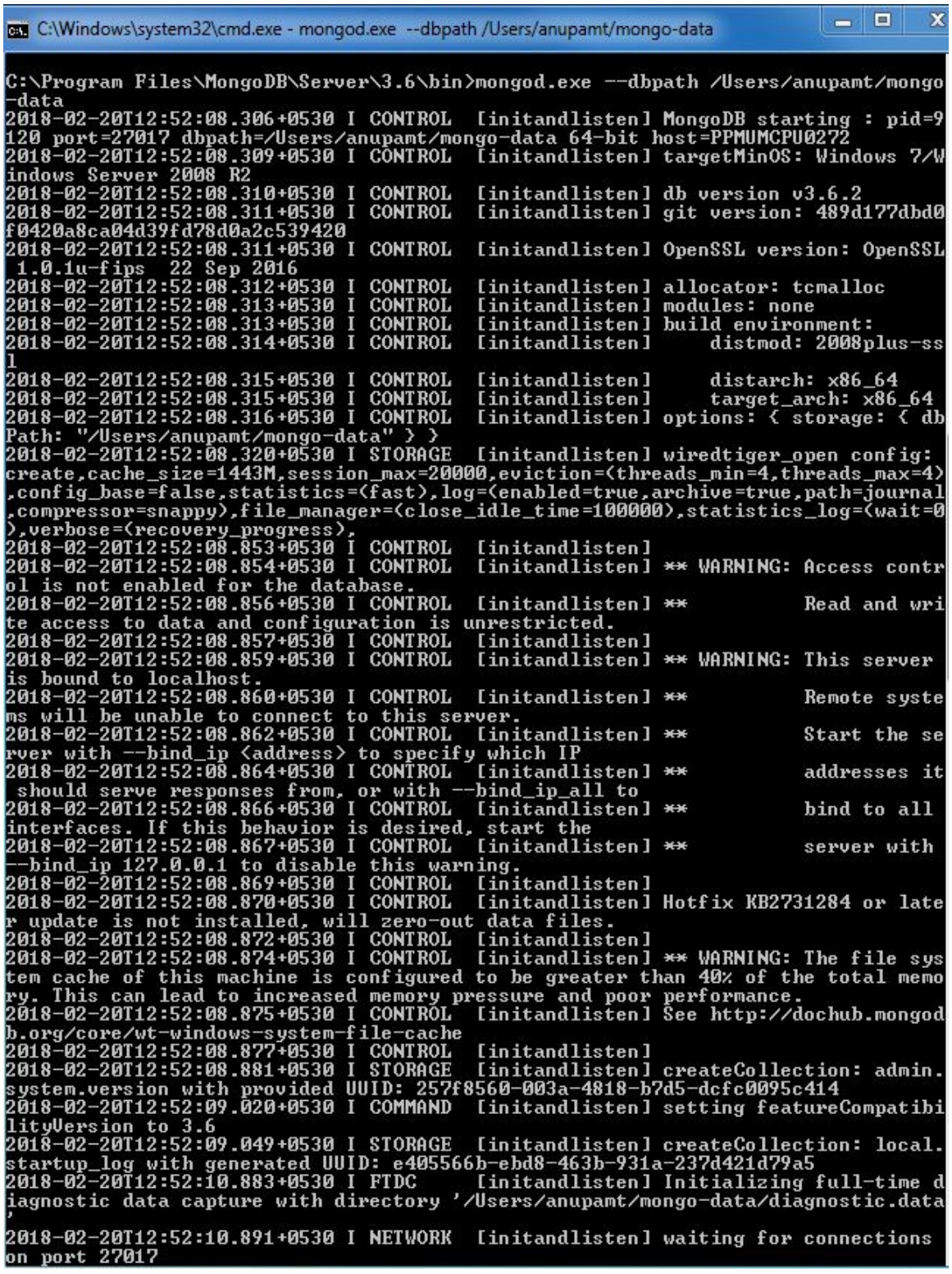
The first executable we'll run is this mongod.exe file. This will start our local MongoDB database. Before we can go ahead and run this `EXE`, there is one more thing we need to do. Over in the generic File Explorer, we need to create a directory where all of our data can be stored. To do this, I'll put mine in my user directory by going to the `C:/Users/Andrew` directory. I'll make a new folder, and I'll call this folder `mongo-data`. Now, the `mongo-data` directory is where all of our data will actually be stored. This is the path that we need to specify when we run the `mongod.exe` command; we need to tell Mongo where to store

the data.

Over in Command Prompt, we can now start this command. I'll run `mongod.exe`, passing in as the `dbpath` argument, the path to that folder we just created. In my case, it's `/Users/Andrew/mongo-data`. Now if your username is different, which it obviously is, or you put the folder in a different directory, you'll need to specify the absolute path to the `mongo-data` folder. Once you have that though, you can start up the server by running the following command:

```
mongod.exe --dbpath /Users/Andrew/mongo-data
```

You'll get a long list of output:



The only thing you need to care about is that, at the very bottom, you should see waiting for connections on port 27017. If you see this, then you are good to go. But now that the server is up, let's

connect to it and issue some commands to create and read some data.

Creating and reading data

To do this, we'll open up a second Command Prompt window and navigate into that same `bin` directory using `cd/Program Files/MongoDB/Server/3.2/bin`. From here, we'll run `mongo.exe`. Note that we're not running the `mongod` command; we're running `mongo.exe`. This will connect to our local MongoDB database, and it will put us in sort of a Command Prompt view of our database. We'll be able to issue various Mongo commands to manipulate the data, kind of like we can run Node from Command Prompt to run various JavaScript statements right inside the console. When we run this, we're going to connect to the database. Over in the first console window, you can see that connection accepted shows up. We do have a new connection. In the first console window now, we can run some commands to create and read data. Now I don't expect you to take away anything from these commands. We'll not talk about the ins and outs of MongoDB just yet. All I want to do is to make sure that when you run them, it works as expected.

To get started, let's create a new Todo from the console. This can be done via `db.Todos`, and on this Todos collection, we'll call the `.insert` method. Also, we'll call `insert` with one argument, an object; this object can have any properties we want to add to the record. For example, I want to set a `text` property. This is the thing I actually need to do. Inside quotes, I can put something. I'll go with `Create new Node course`:

```
db.Todos.insert({text: 'Create new Node course'})
```

Now when I run this command, it will actually make the insert into our database and we should get a `writeResult` object back, with an `nInserted` property set to `1`. This means that one record was inserted.

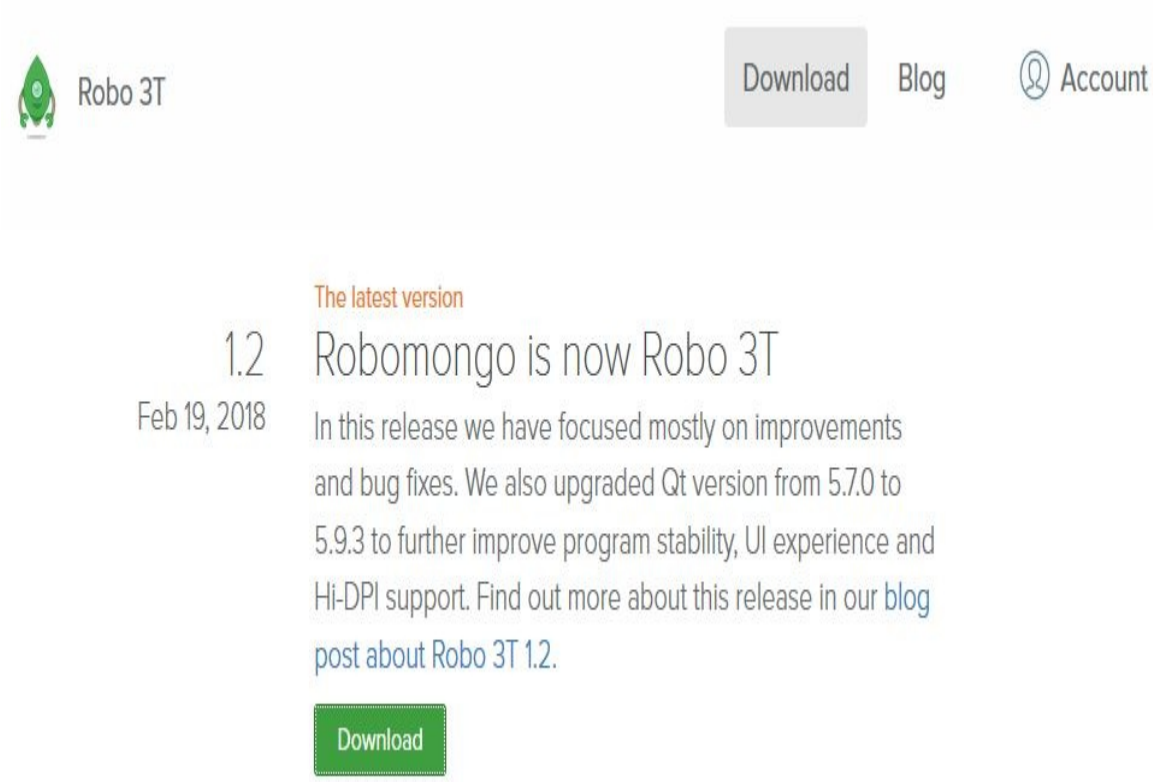
Now that we have one Todo in our database, we can try to fetch it using `db.Todos` once again. This time, instead of calling `insert` to add a record, we'll call `find` with no arguments provided. This will return every single Todo inside of our database:

```
db.Todos.find()
```

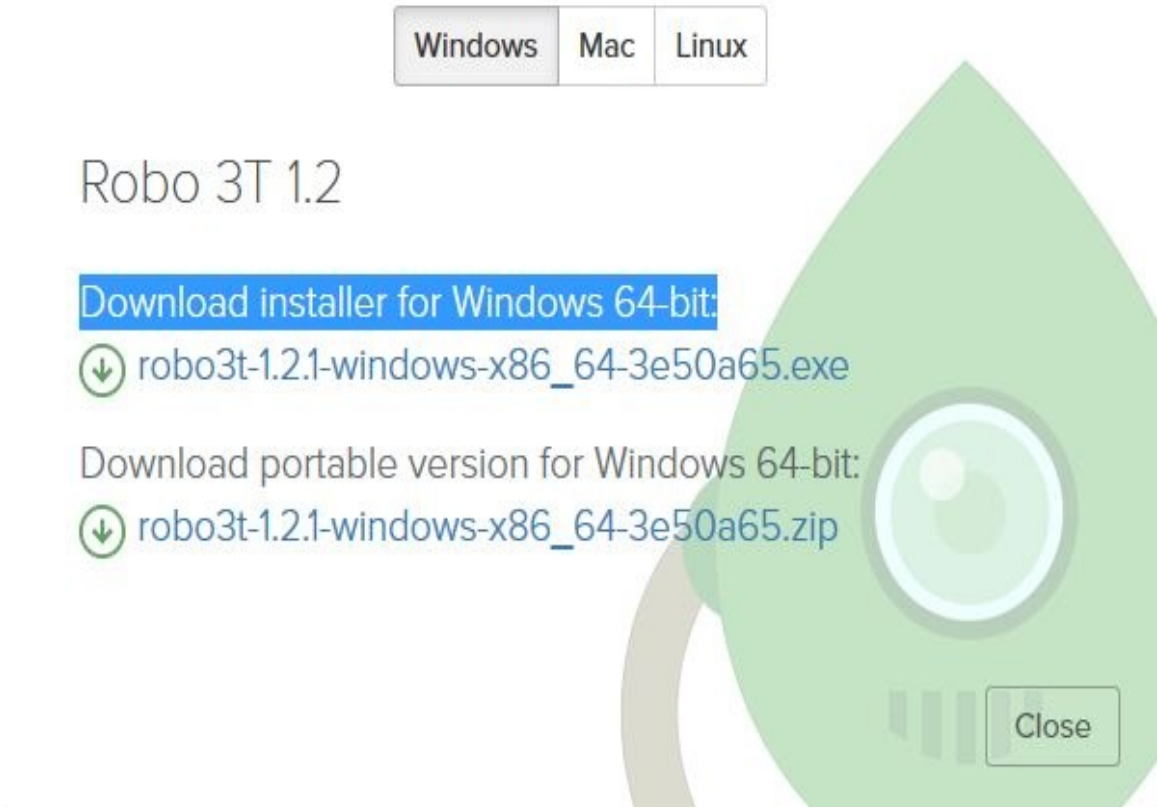
When I run this command, We get an object-looking thing where we have a `text` property set to `Create new Node course`. We also have an `_id` property. The `_id` property is MongoDB's unique identifier, and this is the property that they use to give your document; in this case, a Todo, a unique identifier. We'll be talking more about `_id` and about all of the commands we just ran, a little later. For now, we can close this using *Ctrl + C*. We've successfully disconnected from Mongo, and now we can also close the second Command Prompt window.

Before we move on, there is one more thing I want to do. We'll be installing a program called Robomongo—a GUI for MongoDB. It will let you connect to your local database as well as real databases, which we'll be talking about later. Also, it'll let you view all the data, manipulate it and do anything you could do inside a database GUI. It's really useful; sometimes you just need to dive into a database to see exactly what the data looks like.

In order to get this started, we'll head over to a new tab and go to `robomongo.org`:

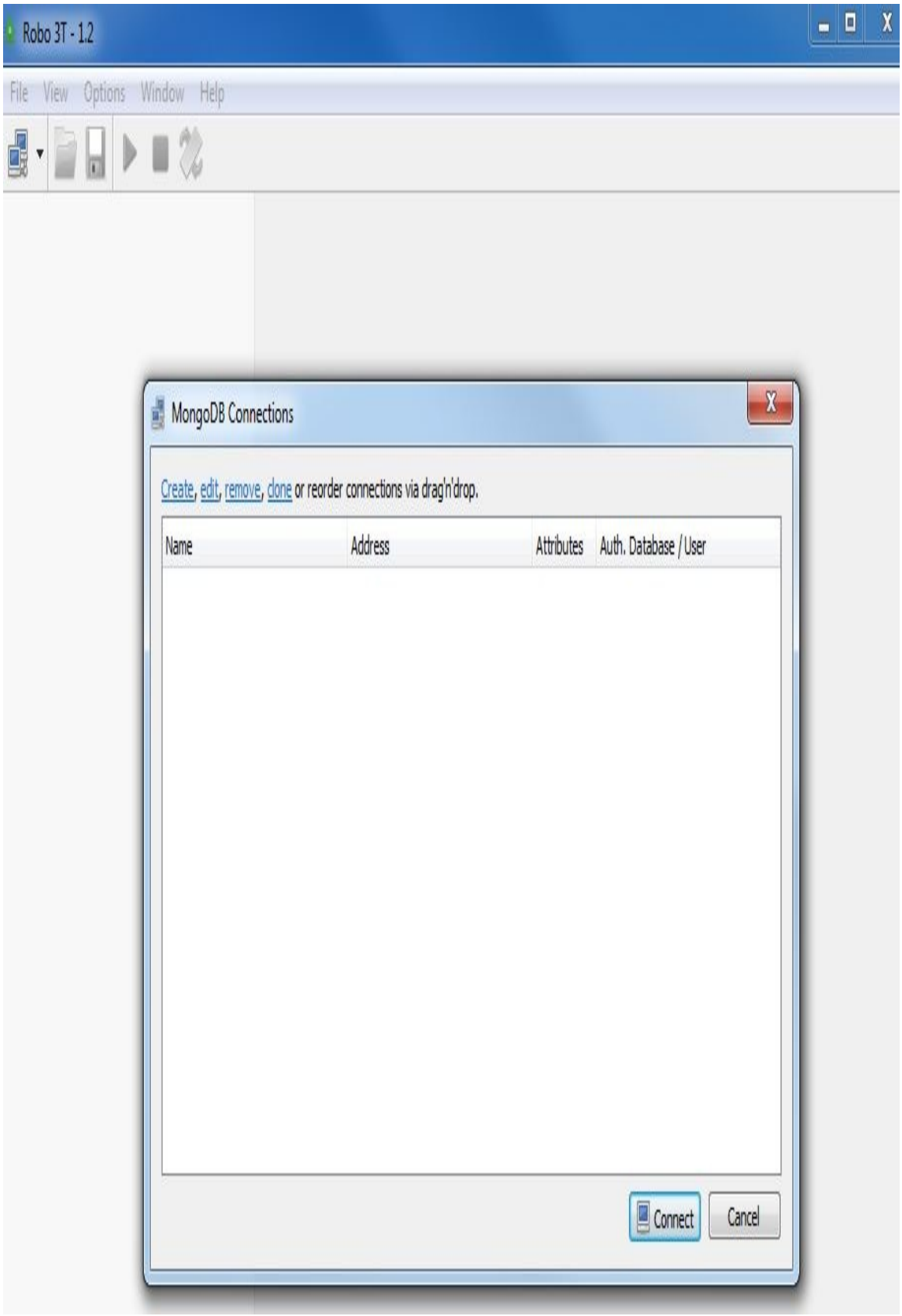


Here we can grab the installer by going to Download. We'll download the latest version, and I'm on Windows. I want the installer, not the portable version, so I'll click on the first link here:



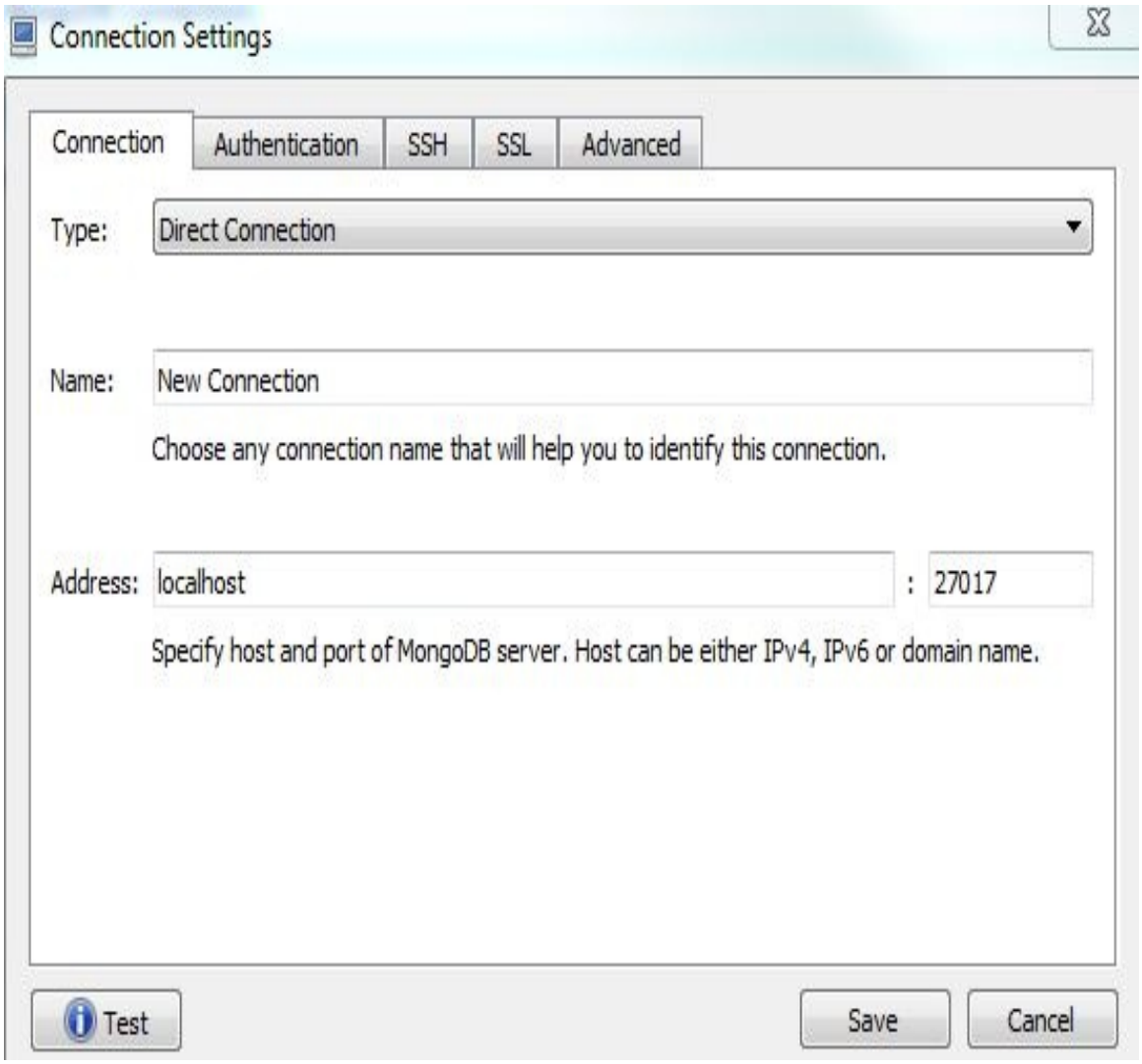
This is going to start a really small download, just 17 MB, and we can click on Next a few times through this one to get Robomongo installed on our machines.

I'll start the process, confirming installation and clicking on Next just a couple of times. There's no need to do anything custom inside the settings. We'll run the installer with all of the default settings applied. Now we can actually run the program by finishing all the steps in the installer. When you run Robomongo, you'll be greeted with a MongoDB Connections screen:


Robo 3T - 1.2
File View Options Window Help
MongoDB Connections
Create, edit, remove, done or reorder connections via drag'n'drop.
Name
Address
Attributes
Auth. Database / User
Connect
Cancel

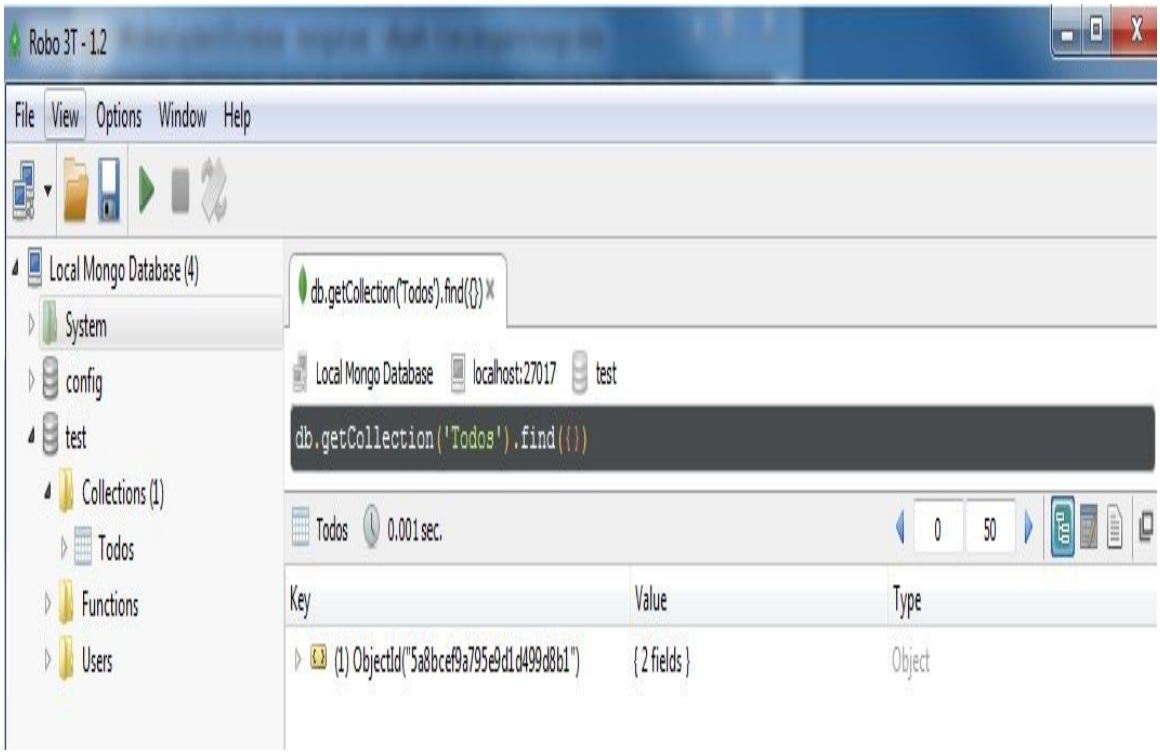
This screen lets you configure all of the connections for Robomongo. You might have a local connection for your local database, and you might have a connection to a real URL where your actual production data is stored. We'll get into all that later.

For now, we'll click on Create. By default, your `localhost` address and your `27017` port do not need to be changed:



All I'm going to do is to change the name so that it's a little easier to identify. I'll go with `Local Mongo Database`. Now, we can save our new connection and actually connect to the database by simply double-clicking on it. When we do that, we get a little tree view of our database. We have this `test` database; this is the one that's created

by default, which we can expand. Then we can expand our `Collections` folder and see the `Todos` collection. This is the collection we created inside the console. I'll right-click on this and go to View Documents. When I view the documents, I actually get to view the individual records:



Here, I see my _id and text properties that have Create new Node course sitting in the above image.

If you are seeing this, then this means that you have a local Mongo server running, and it also means that you've successfully inserted data into it.

Summary

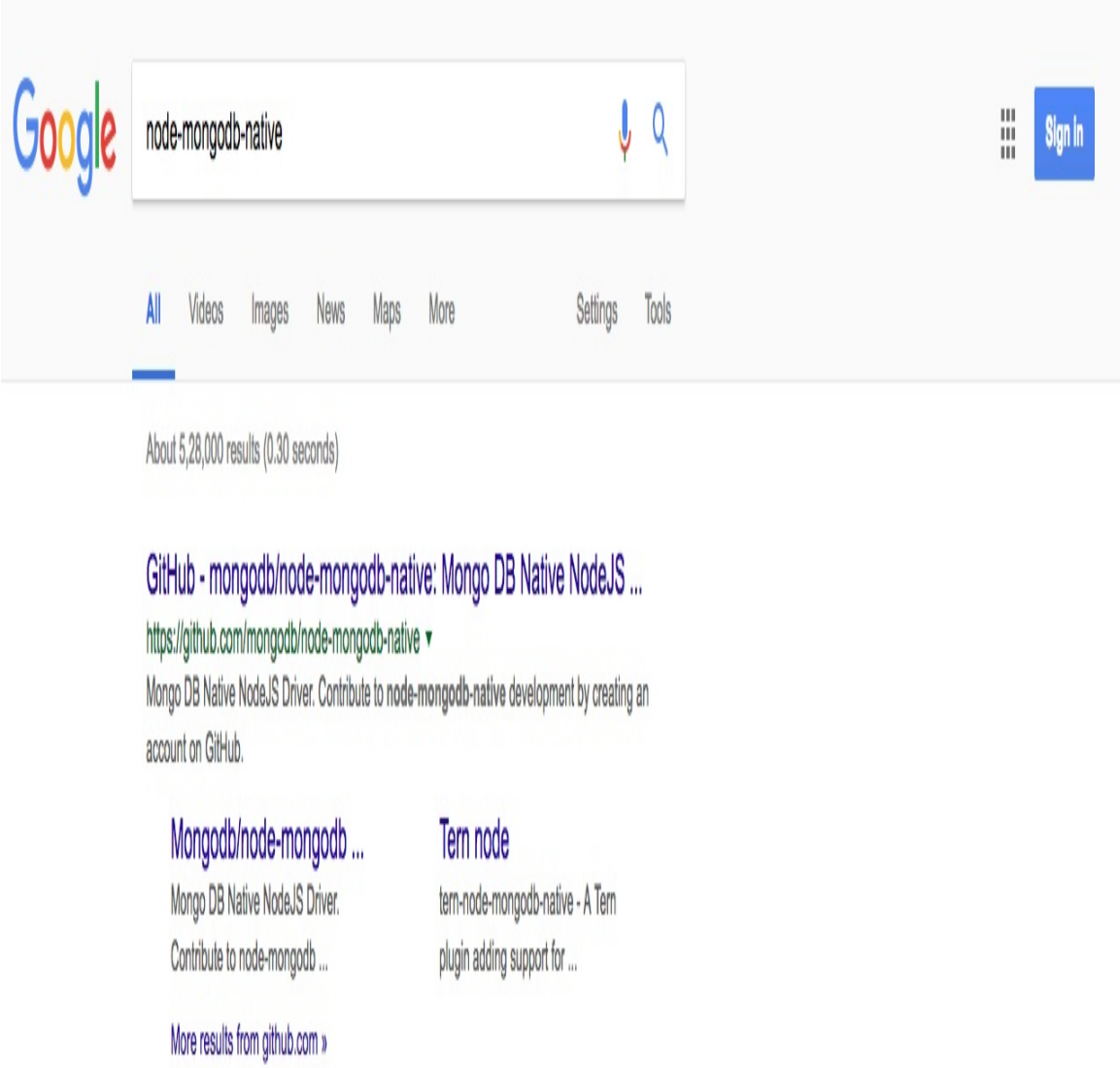
In this chapter, you downloaded and ran the MongoDB database server. This means that we have a local database server we can connect to from our Node application. We also installed Robomongo, which lets us connect to our local database so that we can view and manipulate data. This comes in handy when you're debugging or managing data, or doing anything else with your Mongo database. We'll be using it throughout the book, and you'll begin to see why it's valuable in the later chapters. For now though, you are all set up. You are ready to continue on and start building the Todo API.

MongoDB, Mongoose, and REST APIs – Part 1

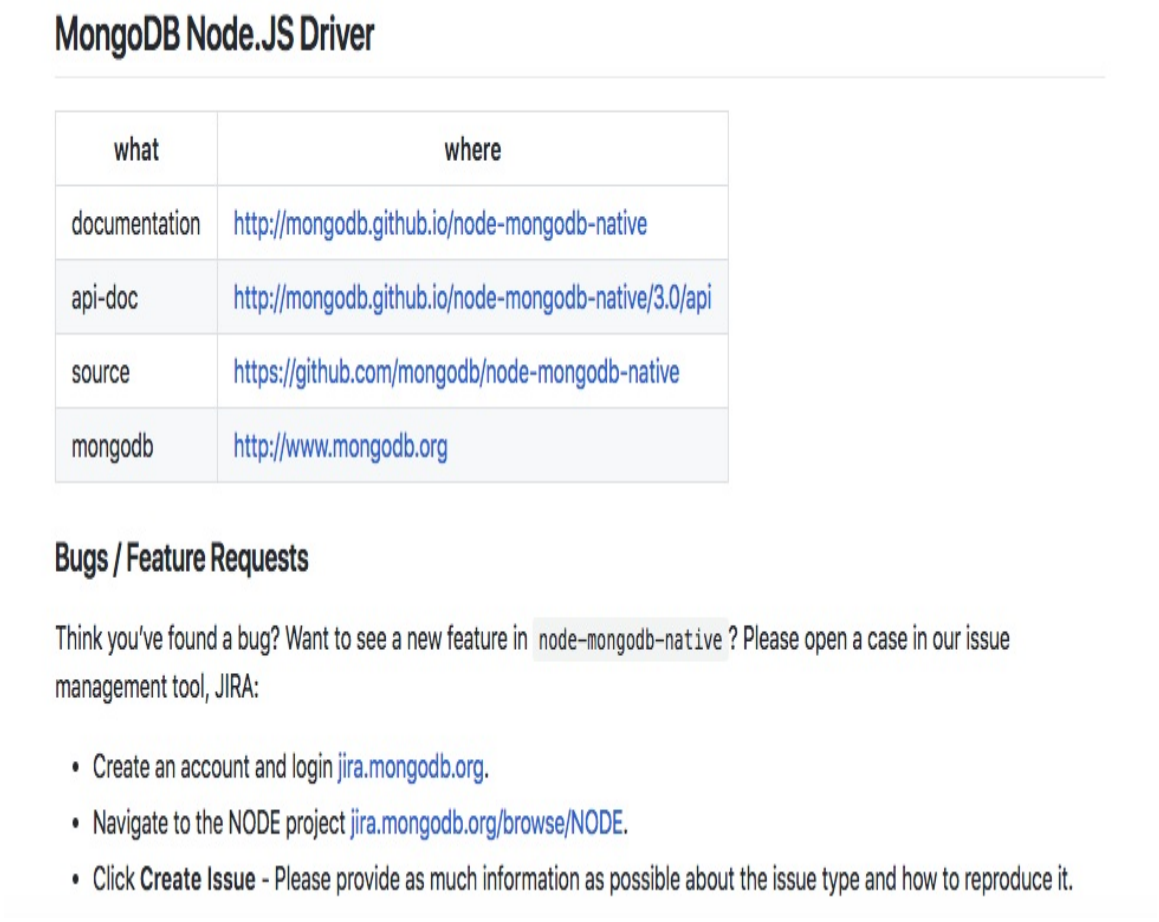
In this chapter, you're going to learn how to connect your Node applications to the MongoDB database you've been running on your local machine. This means that we'll be able to issue database commands right inside of our Node apps to do stuff like insert, update, delete, or read data. This is going to be critical if we're ever going to make that Todo REST API. When someone hits one of our API endpoints, we want to manipulate the database, whether it's reading all of the Todos or adding a new one. Before we can do any of that though, we have to learn the basics.

Connecting to MongoDB and writing data

To connect to our MongoDB database from inside of Node.js, we're going to be using an npm module created by the MongoDB team. It's called node-mongodb-native, but it includes all of the features you'll need to connect to and interact with your database. To get to it, we're going to Google `node-mongodb-native`:



The GitHub repo, which should be the first link, is the one we want —the node-mongodb-native repository—and if we scroll down, we can take a look at a few important links:



MongoDB Node.JS Driver

what	where
documentation	http://mongodb.github.io/node-mongodb-native
api-doc	http://mongodb.github.io/node-mongodb-native/3.0/api
source	https://github.com/mongodb/node-mongodb-native
mongodb	http://www.mongodb.org

Bugs / Feature Requests

Think you've found a bug? Want to see a new feature in `node-mongodb-native`? Please open a case in our issue management tool, JIRA:

- Create an account and login jira.mongodb.org.
- Navigate to the NODE project jira.mongodb.org/browse/NODE.
- Click **Create Issue** - Please provide as much information as possible about the issue type and how to reproduce it.

First up we have documentation, and we also have our api-docs; these are going to be critical as we start exploring the features that we have inside of this library. If we scroll down further on this page, we'll find a ton of examples on how to get started. We'll be going through a lot of this stuff in this chapter, but I do want to make you aware of where you can find other resources because the mongodb-native library has a ton of features. There are entire courses dedicated to MongoDB, and they don't even begin to cover everything that's built-in to this library.

We're going to be focusing on the important and common subset of MongoDB that we need for Node.js apps. To get started, let's go ahead and open up the documentations, which are shown in the preceding image. When you go to the docs page, you have to pick your version. We'll be using version 3.0 of the driver, and there's two important links:

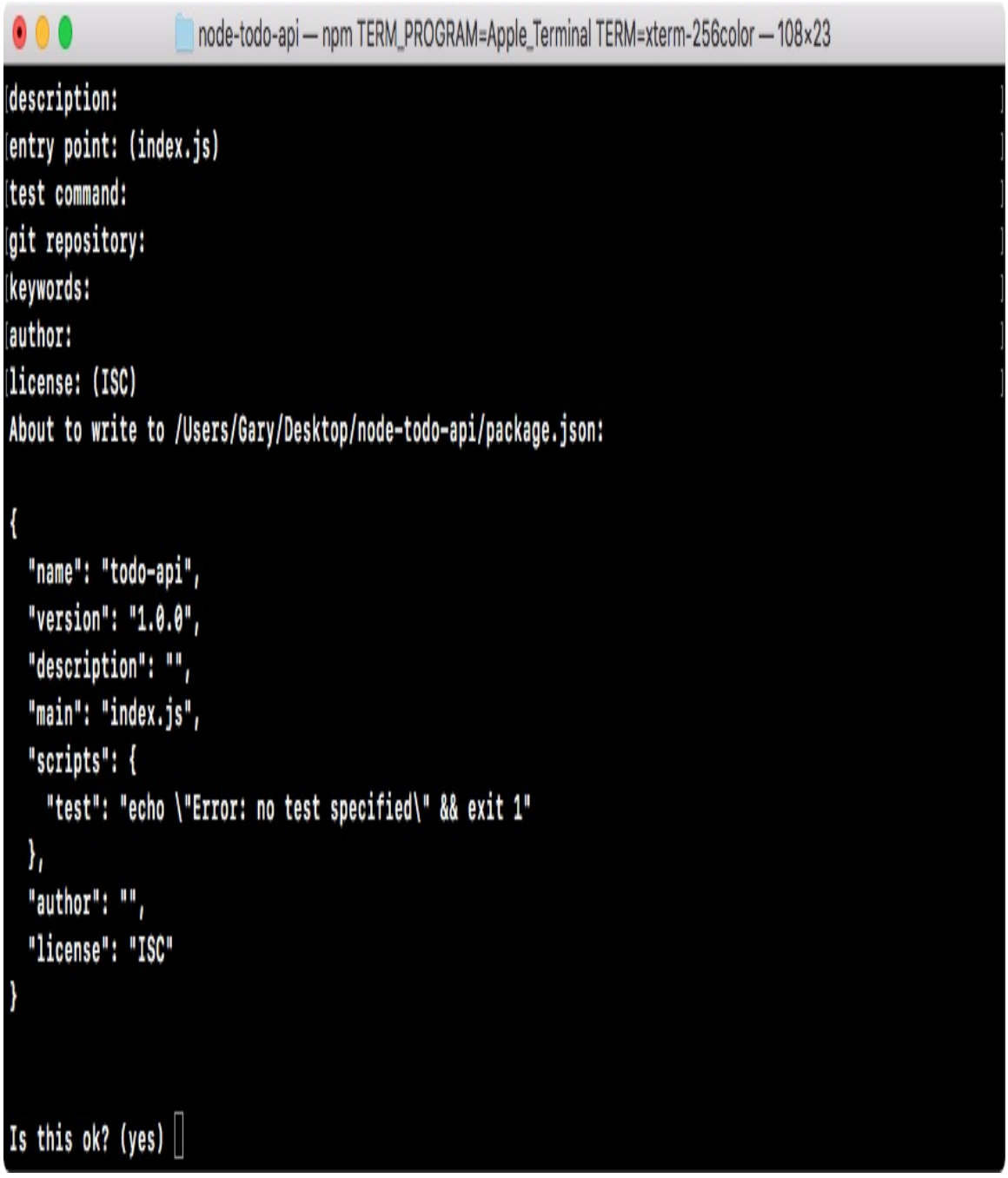
- **The Reference link:** This includes guide-like articles, things to get you started, and other various references.
- **The API link:** This includes the details of every single method available to you when you're working with the library. We'll be exploring some of the methods on this link as we start creating our Node Todo API.

For now though, we can get started by creating a new directory for this project, and then we're going to go ahead and install the MongoDB library and connect to the database we have running. I am going to assume that you have your database running for all the sections in this chapter. I have it running in a separate tab in my Terminal.

If you're on Windows, refer to the instructions in the Windows installation section to start your database if you forget. If you're on a Linux or macOS operating system, use the instructions I have already mentioned, and don't forget to also include that `dbpath` argument, which is essential for booting up your MongoDB server.

Creating a directory for the project

To kick things off, I'm going to make a new folder on the Desktop for the Node API. I'll use `mkdir` to create a new folder called `node-todo-api`. Then, I can go ahead and use `cd` to go into that directory, `cd node-todo-api`. And from here, we're going to run `npm init`, which creates our `package.json` file and lets us install our MongoDB library. Once again, we're going to be using enter to skip through all of the options, using the defaults for each:



Once we get to the end we can confirm our selections, and now our `package.json` file is created. The next thing we're going to do is open up this directory inside of Atom. It's on the Desktop, `node-todo-api`. Next up, inside of the root of the project we're going to create a new folder, and I'm going to call this folder `playground`. Inside of this folder, we'll store various scripts. They're not going to be scripts related to the Todo API; they'll be scripts related to MongoDB, so I

do want to keep them in the folder, but I don't necessarily want them to be part of the app. We'll use the `playground` folder for that, like we have in the past.

In the `playground` folder, let's go ahead and make a new file, and we'll call this file `mongodb-connect.js`. Inside of this file, we're going to get started by loading in the library and connecting to the database. Now in order to do that, we have to install the library. From the Terminal, we can run `npm install` to get that done. The new library name is `mongodb`; all lowercase, no hyphens. Then, we're going to go ahead and specify the version to make sure we're all using the same functionality, `@3.0.2`. This is the most recent version at the time of writing. After the version number, I am going to use the `--save` flag. This is going to save it as a regular dependency, which it already is:

```
npm install mongodb@3.0.2 --save
```

We're going to need this to run the Todo API application.

Connecting the mongodb-connect file to the database

With MongoDB now installed, we can move it to our `mongodb-connect` file and start connecting to the database. The first thing we need to do is pull something out of the library that we just installed, which is the `mongodb` library. What we're looking for is something called the `MongoClient` constructor. The `MongoClient` constructor lets you connect to a Mongo server and issue commands to manipulate the database. Let's go ahead and kick things off by creating a constant called `MongoClient`. We're going to set that equal to `require`, and we're going to require the library we just installed, `mongodb`. From that library, we're going to pull off `MongoClient`:

```
const MongoClient = require('mongodb').MongoClient;
```

With the `MongoClient` now in place, we can call `MongoClient.connect` to connect to the database. This is a method, and it takes two arguments:

- The first argument is a string, and this is going to be the URL where your database lives. Now in a production example, this might be an Amazon Web Services URL or a Heroku URL. In our case, it's going to be a localhost URL. We'll talk about that later.
- The second argument is going to be a callback function. The callback function will fire after the connection has either succeeded or failed, and then we can go ahead and handle

things appropriately. If the connection failed, we'll print a message and stop the program. If it succeeded, we can start manipulating the database.

Adding a string as the first argument

For the first argument in our case, we're going to start off with `mongodb://`. When we connect to a MongoDB database, we want to use the mongodb protocol like this:

```
MongoClient.connect('mongodb://')
```

Next up, it's going to be at localhost since we're running it on our local machine, and we have the port, which we have already explored: `27017`. After the port, we need to use `/` to specify which database we want to connect to. Now, in the previous chapter, we used that test database. This is the default database that MongoDB gives you, but we could go ahead and create a new one. After the `/`, I'm going to call the database `TodoApp`, just like this:

```
MongoClient.connect('mongodb://localhost:27017/TodoApp');
```

Adding the callback function as the second argument

Next up, we can go ahead and provide the callback function. I'm going to use an ES6 arrow (`=>`) function, and we're going to get past two arguments. The first one is going to be an error argument. This may or may not exist; just like we've seen in the past, it'll exist if an error actually happened; otherwise it won't. The second argument is going to be the `client` object. This is what we can use to issue commands to read and write data:

```
MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, client) => {

});
```

Error handling in mongodb-connect

Now, before we write any data, I'm going to go ahead and handle any potential errors that come about. I'll do that using an `if` statement. If there is an error, we're going to print a message to the console, letting whoever is looking at the logs know that we were unable to connect to the database server, `console.log`, then inside of quotes put something like `Unable to connect to MongoDB server`. After the `if` statement, we can go ahead and log out a success message, which will be something like `console.log`. Then, inside of quotes, we'll use `Connected to MongoDB server`:

```
MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, client) => {
  if(err){
    console.log('Unable to connect to MongoDB server');
  }
  console.log('Connected to MongoDB server');
});
```

Now, when you're handling errors like this, the success code is going to run even if the error block runs. What we want to do instead is add a `return` statement right before the `console.log('Unable to connect to MongoDB server');` line.

This `return` statement isn't doing anything fancy. All we're doing is using it to prevent the rest of the function from executing. As soon as you return from a function, the program stops, which means if an error does occur, the message will get logged, the function will stop, and we'll never see this `Connected to MongoDB server` message:

```
if(err) {
```

```
    return console.log('Unable to connect to MongoDB server');
  }
```

An alternative to using the `return` keyword would be to add an `else` clause and put our success code in an `else` clause, but it's unnecessary. We can just use the `return` syntax, which I prefer.

Now, before we run this file, there is one more thing I want to do. At the very bottom of our callback function, we're going to call a method on db. It's called `client.close`:

```
MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, client) => {
  if(err) {
    return console.log('Unable to connect to MongoDB server');
  }
  console.log('Connected to MongoDB server');
  const db = client.db('TodoApp');

  client.close();
});
```

This closes the connection with the MongoDB server. Now that we have this in place, we can actually save the `mongodb-connect` file and run it inside of the Terminal. It doesn't do much yet, but it is indeed going to work.

Running the file in the Terminal

Inside the Terminal, we can run the file using `node playground` as the directory, with the file itself being `mongodb-connect.js`:

```
node playground/mongodb-connect.js
```

When we run this file, we get `Connected to MongoDB server` printing to the screen:

If we head over into the tab where we have the MongoDB server, we can see we got a new connection: connection accepted. As you can see in the following screenshot, that connection was closed down, which is fantastic:

Using the Mongo library we were able to connect, print a message, and disconnect from the server.

Now, you might have noticed that we changed the database name in the `MongoClient.connect` line in Atom, and we never actually did anything to create it. In MongoDB, unlike other database programs, you don't need to create a database before you start using it. If I want to kick up a new database I simply give it a name, something like `Users`.

Now that I have a `Users` database, I can connect to it and I can manipulate it. There is no need to create that database first. I'm going to go ahead and change the database name back to `TodoApp`. If we head into the Robomongo program and connect to our local database, you'll also see that the only database we have is `test`. The `TodoApp` database was never even created, even though we connected to it. Mongo is not going to create the database until we start adding data into it. We can go ahead and do that right now.

Adding data to the database

Inside of Atom, before our call to `db.close`, we're going to insert a new record into a collection. This is going to be the Todo application. We're going to have two collections in this app:

- a `Todos` collection
- a `Users` collection

We can go ahead and start adding some data to the `Todos` collection by calling `db.collection`. The `db.collection` method takes the string name for the collection you want to insert into as its only argument. Now, like the actual database itself, you don't need to create this collection first. You can simply give it a name, like `Todos`, and you can start inserting into it. There is no need to run any command to create it:

```
db.collection('Todos')
```

Next, we're going to use a method available in our collection called `insertOne`. The `insertOne` method lets you insert a new document into your collection. It takes two arguments:

- The first one is going to be an object. This is going to store the various key-value pairs we want to have in our document.
- The second one is going to be a callback function. This callback function will get fired when things either fail or go

well.

You're going to get an error argument, which may or may not exist, and you'll also get the result argument, which is going to be provided if things went well:

```
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, client) => {
  if(err){
    console.log('Unable to connect to MongoDB server');
  }
  console.log('Connected to MongoDB server');
  const db = client.db('TodoApp');
  db.collection('Todos').insertOne({
    text: 'Something to do',
    completed: false
  }, (err, result) => {

  });
  client.close();
});
```

Inside of the error callback function itself, we can add some code to handle the error, and then we'll add some code to print the object to the screen if it was added successfully. First up, let's add an error handler. Much like we have done previously, we're going to check if the error argument exists. If it does, then we'll simply print a message using the `return` keyword to stop the function from executing. Next, we can use `console.log` to print `Unable to insert todo`. The second argument I'm going to pass to the `console.log` is going to be the actual `err` object itself, so if someone's looking at the logs, they can see exactly what went wrong:

```
db.collection('Todos').insertOne({
  text: 'Something to do',
  completed: false
}, (err, result) => {
  if(err){
    return console.log('Unable to insert todo', err);
```

```
  }
```

Next to our `if` statement, we can add our success code. In this case, all we're going to do is pretty-print something to the `console.log` screen, and then I'm going to call `JSON.stringify`, where we're going to go ahead and pass in `result.ops`. The `ops` attribute is going to store all of the docs that were inserted. In this case, we used `insertOne`, so it's just going to be our one document. Then, I can add my other two arguments, which are `undefined` for the filter function, and `2` for the indentation:

```
db.collection('Todos').insertOne({
  text: 'Something to do',
  completed: false
}, (err, result) => {
  if(err){
    return console.log('Unable to insert todo', err);
  }

  console.log(JSON.stringify(result.ops, undefined, 2));
});
```

With this in place, we can now go ahead and execute our file and see what happens. Inside of the Terminal, I'm going to run the following command:

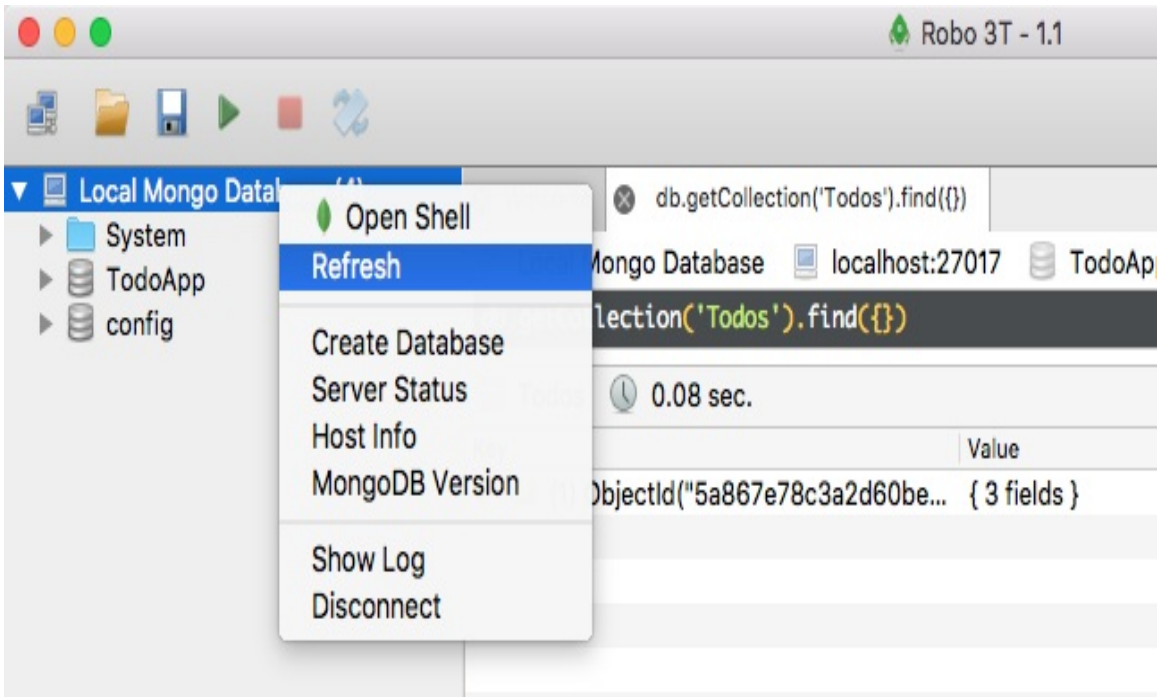
```
node playground/ mongodb-connect.js
```

When I execute the command, we get our success message: `Connected to MongoDB server`. Then, we get an array of documents that were inserted:

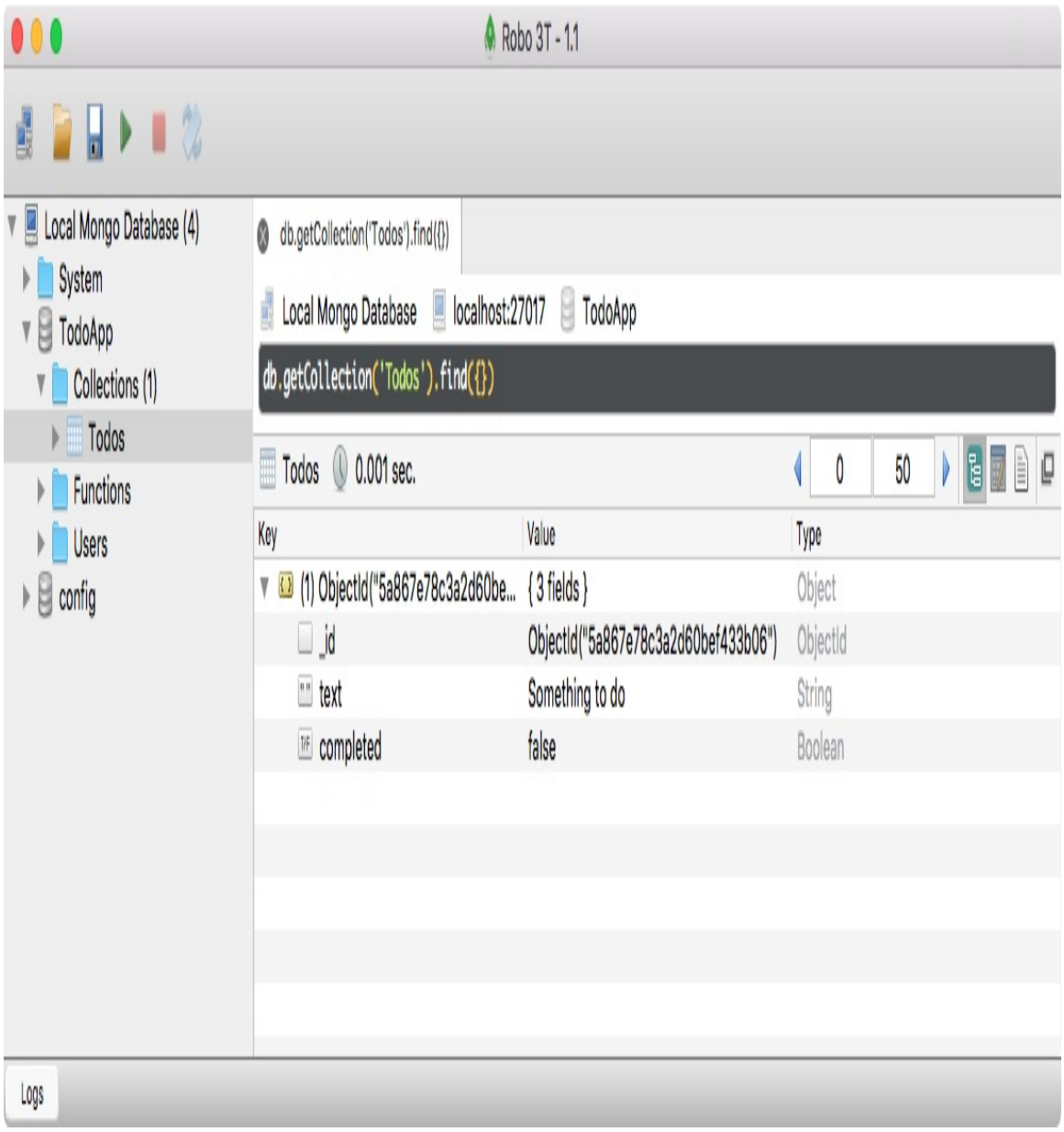
```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
Connected to MongoDB server
[
  {
    "text": "Something to do",
    "completed": false,
    "_id": "5a867e78c3a2d60bef433b06"
  }
]
Gary:node-todo-api Gary$
```

Now as I mentioned, in this case we just inserted one document, and that shows up as shown in the preceding screenshot. We have the `text` property, which gets created by us; we have the `completed` property, which gets created by us; and we have the `_id` property, which gets automatically added by Mongo. The `_id` property is going to be the topic of the following section. We're going to talk in depth about what it is, why it exists and why it's awesome.

For now, we're going to go ahead and just note that it's a unique identifier. It's an ID given to just this document. That is all it takes to insert a document into your MongoDB database using Node.js. We can view this document inside of Robomongo. I'm going to right-click the connection, and click Refresh:



This reveals our brand new `TodoApp` database. If we open that up, we get our `Collections` list. We can then go into the `Collections`, view the documents, and what do we get? We get our one Todo item. If we expand it, we can see we have our _id, we have our text property, and we have our completed Boolean:



In this case, the Todo is not completed, so the completed value is false. Now, what I want you to do is add a new record into a collection. This is going to be your challenge for the section.

Adding a new record into a collection

Inside of Atom, what I'd like you to do is take the code all the way from `db.collection` down to the bottom of our callback, and comment it out. Then, we're going to go ahead and add something following it. Right previous `db.close()`, you're going to type `Insert new doc into the Users collection`. This doc is going to have a few properties. I want you to give it a `name` property; set that equal to your name. Then, we're going to give it an `age` property, and last but not least we can give it a `location` string. I want you to insert that doc using `insertOne`. You're going to need to pass in the new collection name into the collection method. Then, further down, you're going to add some error-handling code, and you're going to print the ops to the screen. Once you rerun the file, you should be able to view your record in the Terminal and you should be able to refresh things. Over in Robomongo, you should see the new Users collection, and you should see your user with the name, age, and location you specified.

Hopefully, you were able to insert a new document into the Users collection. What you needed to do in order to get this done is call `db.collection` so we can access the collection we want to insert into, which in this case is `Users`:

```
//Insert new doc into Users(name, age, location)
db.collection('Users')
```

Next up, we have to call a method to manipulate the `Users` collection. We want to insert a new document, so we're going to use `insertOne`, just like we did in the previous sub-section. We're going to pass our two arguments into `insertOne`. The first one is the document to insert.

We're going to give it a `name` property; I'll set that equal to `Andrew`. Then, we can go ahead and set the `age` equal to something like `25`. Lastly, we'll set the `location` equal to my current location, `Philadelphia`:

```
//Insert new doc into Users(name, age, location)
db.collection('Users').insertOne({
  name: 'Andrew',
  age: 25,
  location: 'Philadelphia'
}
```

The next argument we want it to pass in is our callback function, which is going to get called with the error object as well as the results. Inside of the callback function itself, we're going to first handle the error. If there was an error, we're going to go ahead and log it to the screen. I'm going to return `console.log`, and then we can put the message: `Unable to insert user`. Then, I'll add the error argument as the second argument for `console.log`. Next up, we can add our success case code. If things go well, all I'm going to do is use `console.log` to print `result.ops` to the screen. This is going to show us all of the records that were inserted:

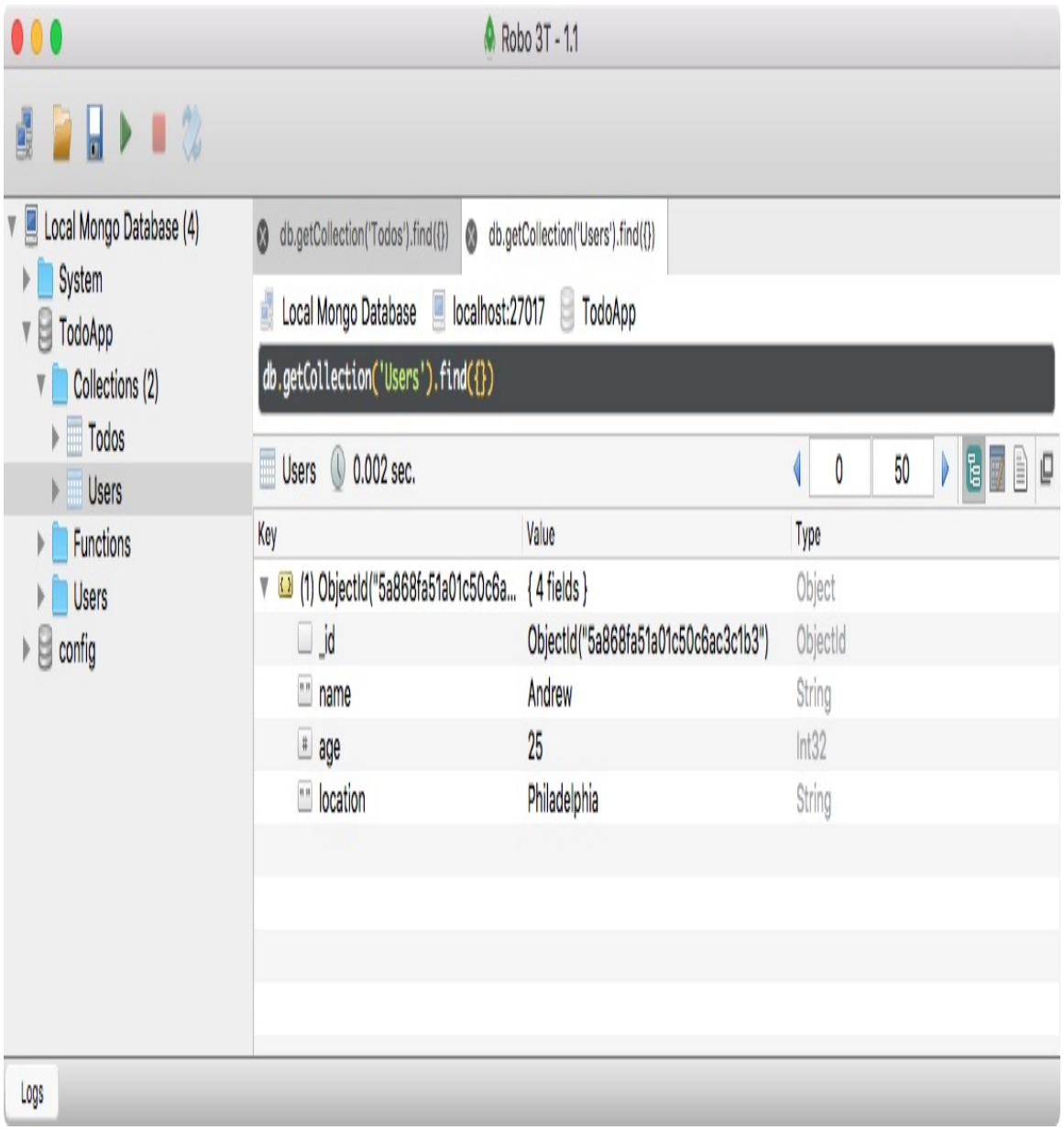
```
//Insert new doc into Users(name, age, location)
db.collection('Users').insertOne({
  name: 'Andrew',
  age: 25,
  location: 'Philadelphia'
}, (err, result) => {
  if(err) {
    return console.log('Unable to insert user', err);
  }
  console.log(result.ops);
});
```

We can now go ahead and rerun the file inside of the Terminal using the *up* arrow key and the *enter* key:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
Connected to MongoDB server
[ { name: 'Andrew',
    age: 25,
    location: 'Philadelphia',
    _id: 5a868fa51a01c50c6ac3c1b3 } ]
Gary:node-todo-api Gary$
```

We get our array of inserted documents, and we just have one. The `name`, `age`, and `location` properties all come from us, and the `_id` property comes from MongoDB.

Next up, I want you to verify that it was indeed inserted by viewing it in Robomongo. In general, when you add a new collection or a new database, you can just right-click the connection itself, click Refresh, and then you should be able to see everything that was added:



As shown in the preceding screenshot, we have our Users collection. I can view the documents for Users. We get our one document with the name set to Andrew, age set to 25, and location set to Philadelphia. With this in place, we are now done. We've been able to connect to our MongoDB database using Node.js, and we've also learned how to insert documents using this mongo-native library. In the next section, we're going to take an in-depth look at ObjectIds, exploring exactly what they are and why they're useful.

The ObjectId

Now that you have inserted some documents into your MongoDB collections, I want to take a moment to talk about the `_id` property in the context of MongoDB because it's a little different than the IDs that you're probably used to if you've used other database systems, like Postgres or MySQL.

The _id property in the context of MongoDB

To kick off our discussion of the `_id` property, let's go ahead and rerun the `mongodb-connect` file. This is going to insert a new document into the Users collection, like we've defined in the `db.collection` line. I'm going to go ahead and do that by running the file through the node. It's in the `playground` folder, and the file itself is called `mongodb-connect.js`:

```
node playground/mongodb-connect.js
```

I'm going to run the command, and we're going to print out the document that got inserted:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
Connected to MongoDB server
[ { name: 'Andrew',
    age: 25,
    location: 'Philadelphia',
    _id: 5a868fa51a01c50c6ac3c1b3 } ]
Gary:node-todo-api Gary$
```

As we've seen in the past, we get our three attributes as well as the one added by Mongo.

The first thing you'll notice about this is that it is not an auto incrementing integer, kind of like it is for Postgres or MySQL, where the first record has an ID of 1 and the second one has an ID of 2. Mongo does not use this approach. Mongo was designed to

scale out really easily. Scaling out means that you can add on more database servers to handle that extra load.

Imagine you have a web app that gets about 200 users a day and your current servers are ready for that traffic. Then, you get picked up by some news outlet and 10,000 people flood your site. With MongoDB, it's really easy to kick up new database servers to handle that extra load. When we use a randomly generated ID, we don't need to constantly communicate with the other database servers to check what the highest incrementing value is. Is it 7? Is it 17? It doesn't really matter; we're simply going to generate a new random ObjectId and use that for the document's unique identifier.

Now, the ObjectId itself is made up of a few different things. It's a 12-byte value. The first four bytes are a timestamp; we'll talk about that later. That means that we have a timestamp built into the data that refers to the moment in time the ID was created. This means that in our documents, we don't need to have a `createdAt` field; it's already encoded in the ID.

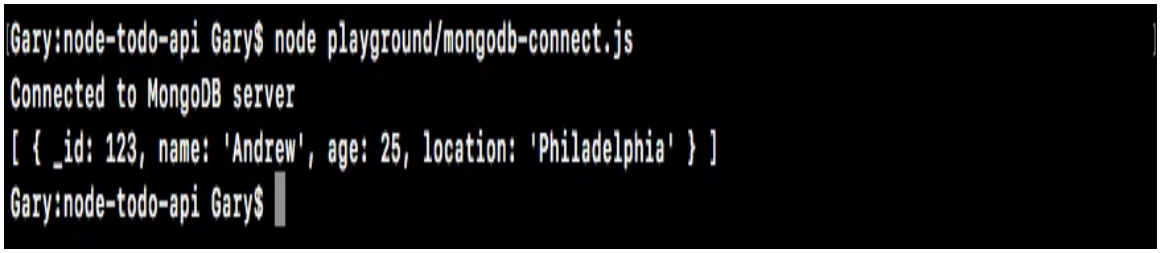
The next three bytes are machine identifiers. This means that if two computers generate ObjectIds, their machine ID is going to be different, and this is going to ensure that the ID is unique. Next up, we have two bytes, the process ID, which is just another way to create a unique identifier. Last up, we have a 3-byte counter. This is similar to what MySQL would do. This is only 3 bytes of the ID. As we have already mentioned, we have a timestamp which is going to be unique; a machine identifier; a process ID; and lastly, just a random value. That is what makes up an ObjectId.

The ObjectId is the default value for `_id`. If nothing is provided, you can indeed do whatever you like with that property. For example, inside of the `mongodb-connect` file, I can specify an `_id` property. I'm going to give it a value, so let's go with `123`; add a comma at the end; and that is perfectly legal:

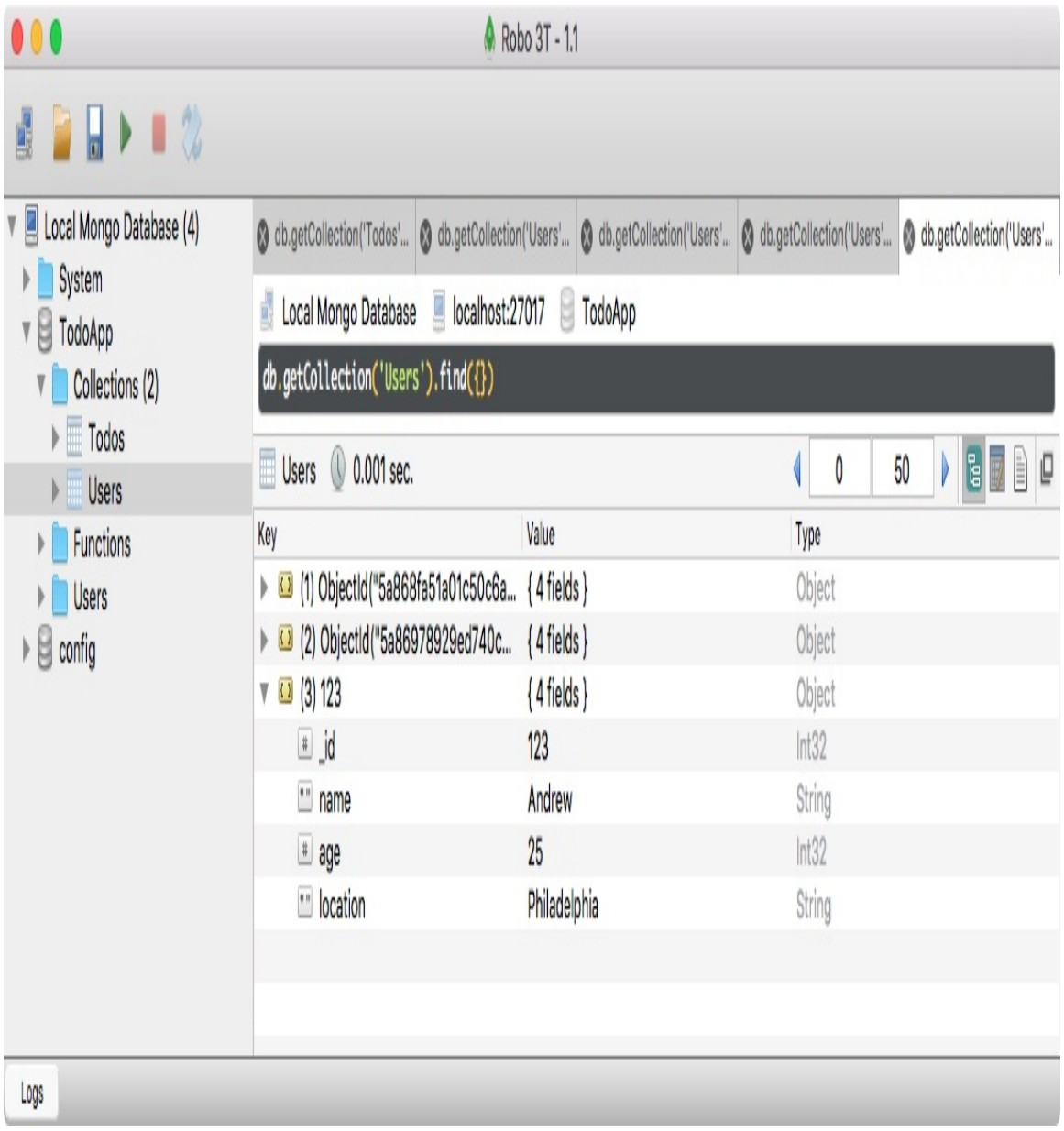
```
db.collection('Users').insertOne({
```

```
  _id: 123,
  name: 'Andrew',
  age: 25,
  location: 'Philadelphia'
}
```

We can save the file, and rerun the script using the *up* arrow key and the *enter* key:



We get our record, where the `_id` property is `123`. The `ObjectId` is the default way MongoDB creates IDs, but you can do anything you like for ID creation. Inside of Robomongo, we can give our Users collection a refresh, and we get our documents:



We have the one we created in the previous section and the two we just made now, all with a unique identifier. This is why unique IDs are really important. In this example, we have three properties: name, age and location, and they're the same for all the records. This is a reasonable thing to do. Imagine two people need to do the same thing, like buy groceries. That string alone is not going to be enough to uniquely identify a Todo. ObjectIds, on the other hand, are going to be unique, and that is what we're going to use to associated things like Todos with things like `Users`.

Next up, I want to take a look at some things we can do with the ID inside of our code. As I mentioned earlier, a timestamp is embedded inside of here, and we can actually pull that out. Inside of Atom, what we're going to do is remove the `_id` property. The timestamp is only going to be available when you're using the `ObjectId`. Then, inside of our callback, we can go ahead and print the timestamp to the screen.

```
db.collection('Users').insertOne({
  name: 'Andrew',
  age: 25,
  location: 'Philadelphia'
}, (err, result) => {
  if(err) {
    return console.log('Unable to insert user', err);
  }

  console.log(result.ops);
});
```

If you remember, `result.ops` is an array of all the documents that got inserted. We're only inserting one, so I'm going to access the first item in the array, and then we're going to access the `_id` property. This is going to do exactly what you might think:

```
console.log(result.ops[0]._id);
```

If we save the file and rerun the script over from the Terminal, all we get is the `ObjectId` printing to the screen:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
Connected to MongoDB server
5a8698e47bcb000cb63cb05a
Gary:node-todo-api Gary$
```

Now though, we can call a method on the `_id` property.

Calling the .getTimestamp function

What we're going to call is `.getTimestamp`. The `getTimestamp` is a function, but it doesn't take any arguments. It simply returns the timestamp that the ObjectId was created at:

```
console.log(result.ops[0]._id.getTimestamp());
```

Now, if we go ahead and rerun our program, we get a timestamp:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
Connected to MongoDB server
2018-02-16T08:41:27.000Z
Gary:node-todo-api Gary$
```

In the preceding screenshot, I can see that the ObjectId was created on February 16th 2016 at 08:41 Z, so this timestamp is indeed correct. This is a fantastic way to figure out exactly when a document was created.

Now, we don't have to rely on MongoDB to create our ObjectIds. Inside of the MongoDB library, they actually give us a function we can run to make an ObjectId whenever we like. For the moment, let's go ahead and comment out our call to insert one.

At the very top of the file, we're going to change our import statement to load in something new off of MongoDB, and we're going to do this using an ES6 feature known as object destructuring. Let's take a quick second to talk about that before we actually go

ahead and use it.

Using object destructuring ES6

Object destructuring lets you pull out properties from an object in order to create variables. This means that if we have an object called `user` and it's set equal to an object with a `name` property set to `andrew` and an age property set to `25`, as shown in the following code:

```
const MongoClient = require('mongodb').MongoClient;

var user = {name: 'andrew', age: 25};
```

We can easily pull out one of these into a variable. Let's say, for example, we want to grab name and create a `name` variable. To do that using object destructuring in ES6, we're going to make a variable and then we're going to wrap it inside of curly braces. We're going to provide the name we want to pull out; this is also going to be the variable name. Then, we're going to set it equal to whatever object we want to destructure. In this case, that is the `user` object:

```
var user = {name: 'andrew', age: 25};
var {name} = user;
```

We have successfully destructured the `user` object, pulling off the `name` property, creating a new `name` variable, and setting it equal to whatever the value is. This means I can use the `console.log` statement to print `name` to the screen:

```
var user = {name: 'andrew', age: 25};
var {name} = user;
console.log(name);
```

I'm going to rerun the script and we get `andrew`, which is exactly what you'd expect because that is the value of the `name` property:

ES6 destructuring is a fantastic way to make new variables from an object's properties. I'm going to go ahead and delete this example, and at the top of the code, we're going to change our `require` statement so that it uses destructuring.

Before we add anything new, let's go ahead and take the MongoClient statement and switch it to destructuring; then, we'll worry about grabbing that new thing that's going to let us make ObjectIds. I'm going to copy and paste the line and comment out the old one so we have it for reference.

```
// const MongoClient = require('mongodb').MongoClient;
const MongoClient = require('mongodb').MongoClient;
```

What we're going to do is remove our `.MongoClient` call after `require`. There's no need to pull off that attribute because we're going to be using destructuring instead. That means over here we can use destructuring, which requires us to add our curly braces, and we can pull off any property from the MongoDB library.

```
const {MongoClient} = require('mongodb');
```

In this case, the only property we had was `MongoClient`. This creates a variable called `MongoClient`, setting it equal to the `MongoClient` property of `require('mongodb')`, which is exactly what we did in the previous `require` statement.

Creating a new instance of objectID

Now that we have some destructuring in place, we can easily pull more things off of MongoDB. We can add a comma and specify something else we want to pull off. In this case, we're going to pull off uppercase, `ObjectID`.

```
const {MongoClient, ObjectID} = require('mongodb');
```

This `ObjectID` constructor function lets us make new ObjectIds on the fly. We can do anything we like with them. Even if we're not using MongoDB as our database, there is some value in creating and using ObjectIds to uniquely identify things. Next, we can make a new ObjectId by first creating a variable. I'll call it `obj`, and we'll set it equal to `new ObjectID`, calling it as a function:

```
const {MongoClient, ObjectID} = require('mongodb');

var obj = new ObjectID();
```

Using the `new` keyword, we can create a new instance of `ObjectID`. Next up, we can go ahead and log that to the screen using `console.log(obj)`. This is a regular ObjectId:

```
console.log(obj);
```

If we rerun the file over from the Terminal, we get exactly what you'd expect:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
5a869c6a8353400cd9161760
Connected to MongoDB server
Gary:node-todo-api Gary$
```

We get an ObjectId-looking thing. If I rerun it again, we get a new one; they are both unique:

```
Gary:node-todo-api Gary$ node playground/mongodb-connect.js
5a869cbe9c794c0ce0597329
Connected to MongoDB server
Gary:node-todo-api Gary$
```

Using this technique, we can incorporate ObjectIds anywhere we like. We could even generate our own, setting them as the `_id` property for our documents, although I find it much easier to let MongoDB handle that heavy lifting for us. I'm going to go ahead and remove the following two lines since we won't actually be using this code in the script:

```
var obj = new ObjectID();
console.log(obj);
```

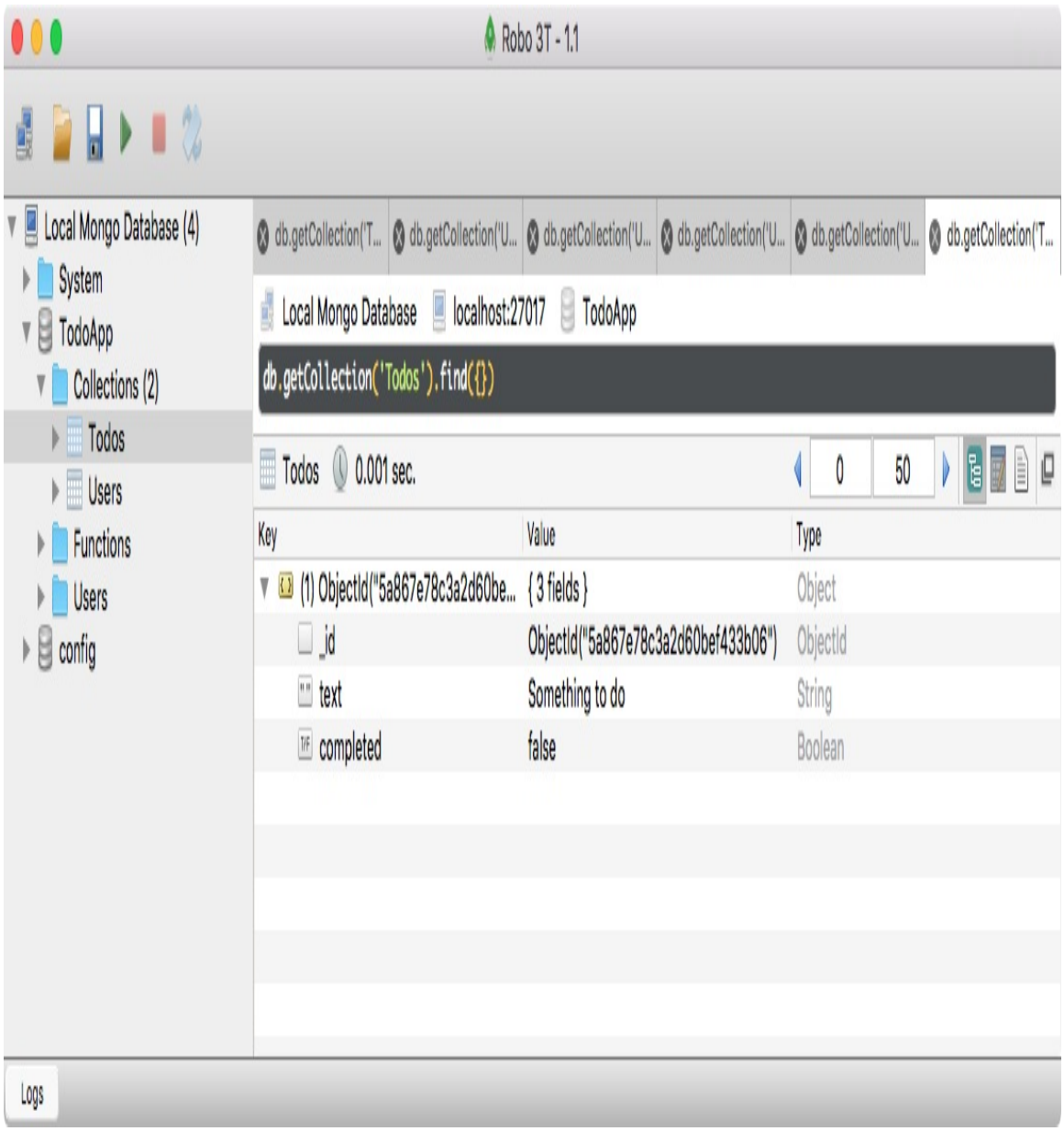
We have learned a bit about ObjectIds, what they are, and why they're useful. In the following sections, we're going to be taking a look at other ways we can work with MongoDB. We'll learn how to read, remove, and update our documents.

Fetching data

Now that you know how to insert data into your database, let's go ahead and talk about how we can fetch data out of it. We're going to be using this technique in the Todo API. People are going to want to populate a list of all the Todo items they need, and they might want to fetch the details about an individual Todo item. All of this is going to require that we can query the MongoDB database.

Fetching todos in Robomongo file

Now, we're going to create a new file based off of `mongodb-connect`. In this new file, instead of inserting records, we'll fetch records from the database. I'm going to create a duplicate, calling this new file `mongodb-find`, because `find` is the method we're going to use to query that database. Next, we can go ahead and remove all of the commented-out code that currently inserts records. Let's get started by trying to fetch all of the Todos out of our Todos collection. Now, if I head over to Robomongo and open up the `Todos` collection, we have just one record:

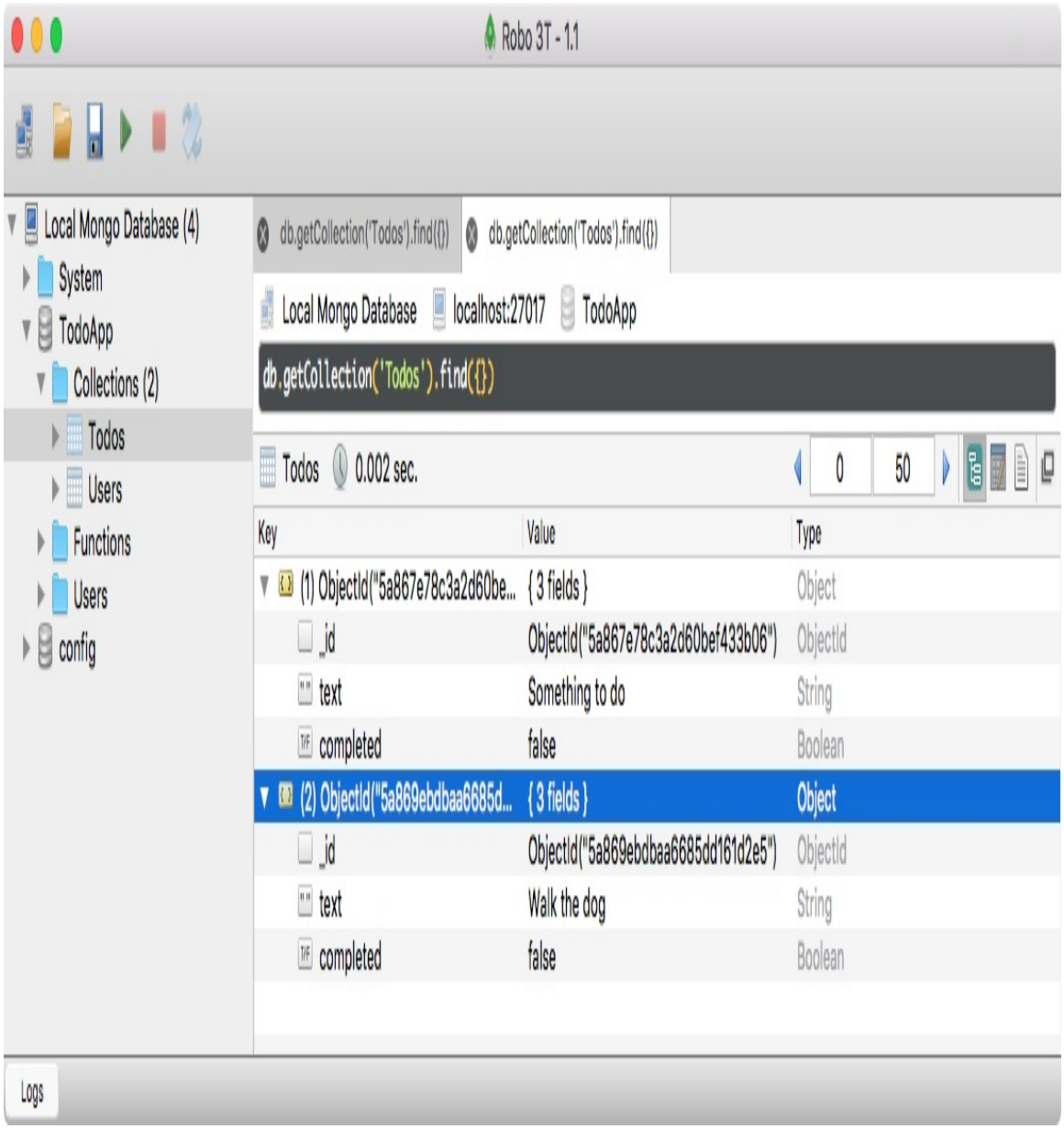


In order to make this querying a little more interesting, we're going to go ahead and add a second one. Right in the Robomongo window, I can click Insert Document. Robomongo can delete, insert, update, and read all of your documents, and this makes it a fantastic tool for debugging. We can add a new document on the fly, with a `text` property equal to `Walk the dog`, and we can also tack on a `completed` value. I'm going to set `completed` equal to `false`:

```
{
  text : "Walk the dog",
```

```
  completed : false
}
```

Now by default, we're not going to provide an `_id` prop. This is going to let MongoDB automatically generate that ObjectId, and right here we have our two Todos:



With this in place, let's go ahead and run our first query inside of Atom.

The find method

In Atom, what we're going to do is access the collection, just like we did inside of the `mongodb-connect` file using `db.collection`, passing in the collection name as the string. This collection is going to be the `Todos` collection. Now, we're going to go ahead and use a method available on collections called `find`. By default, we can call `find` with no arguments:

```
db.collection('Todos').find();
```

This means we're not providing a query, so we're not saying we want to fetch all `Todos` that are completed or not completed. We're just saying we want to fetch all `Todos`: everything, regardless of its values. Now, calling find is only the first step. `find` returns a MongoDB cursor, and this cursor is not the actual documents themselves. There could be a couple of thousand, and that would be really inefficient. It's actually a pointer to those documents, and the cursor has a ton of methods. We can use those methods to get our documents.

One of the most common cursor methods we're going to be using is `.toArray`. It does exactly what you think it does. Instead of having a cursor, we have an array of the documents. This means we have an array of objects. They have ID properties, text properties, and completed properties. This `toArray` method gets us exactly what we want back, which is the documents. `toArray` returns a promise. This means we can tack on a `then` call, we can add our callback, and when things go right, we can do something like print those documents to the screen.

```
db.collection('Todos').find().toArray().then((docs) => {

});
```

We're going to get the documents as the first and only argument here, and we can also add an error handler. We'll get passed an error argument, and we can simply print something to the screen like `console.log(Unable to fetch todos)`; as the second argument, we'll pass in the `err` object:

```
db.collection('Todos').find().toArray().then((docs) => {

}, (err) => {
  console.log('Unable to fetch todos', err);
});
```

Now, for the success case, what we're going to do is print the documents to the screen. I'm going to go ahead and use `console.log` to print a little message, `Todos`, and then I'll call `console.log` again. This time, we'll be using the `JSON.stringify` technique. I'll be passing in the documents, `undefined` for our filter function and `2` for our spacing.

```
  db.collection('Todos').find().toArray().then((docs) => {
    console.log('Todos');
    console.log(JSON.stringify(docs, undefined, 2));
  }, (err) => {
    console.log('Unable to fetch todos', err);
  });
```

We now have a script that is capable of fetching the documents, converting them into an array, and printing them to the screen. Now, for the time being, I'm going to comment out the `db.close` method. Currently, that would interfere with our previous bit of code. Our final code would look as follows:

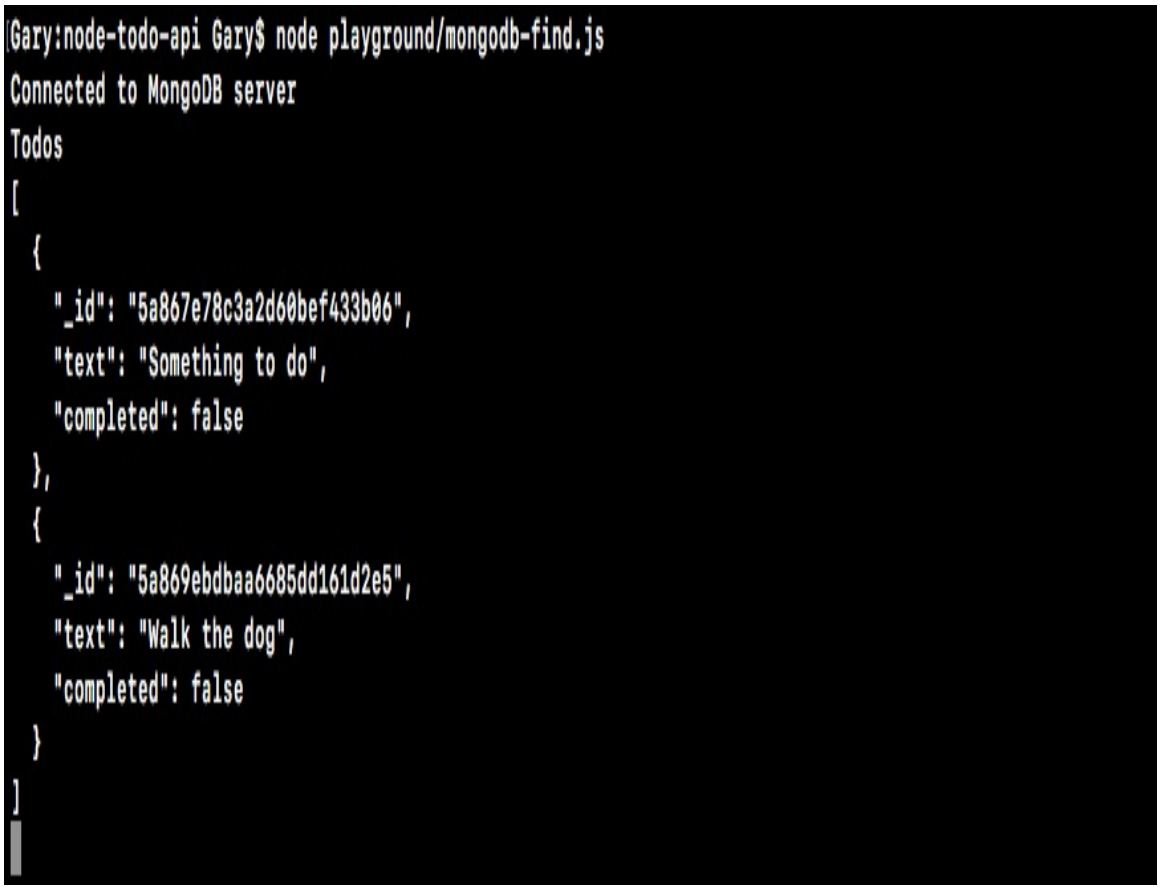
```
//const MongoClient = require('mongodb').MongoClient;
```

```
const {MongoClient, ObjectID} = require('mongodb');

MongoClient.connect('mongodb://localhost:27017/TodoApp', (err, client) => {
  if(err){
    console.log('Unable to connect to MongoDB server');
  }
  console.log('Connected to MongoDB server');
  const db = client.db('TodoApp');

  db.collection('Todos').find().toArray().then((docs) => {
    console.log('Todos');
    console.log(JSON.stringify(docs, undefined, 2));
  }, (err) => {
    console.log('Unable to fetch todos', err);
  });
  //client.close();
});
```

Save the file and run it from the Terminal. Inside of the Terminal, I'm going to go ahead and run our script. Obviously, since we connected to the database with Robomongo, it is running somewhere; it's running in this other tab. In the other tab, I can run the script. We're going to run it through `node`; it's in the `playground` folder, and the file itself is called `mongodb-find.js`:

```
node playground/mongodb-find.js
```

When I execute this file, we're going to get our results:

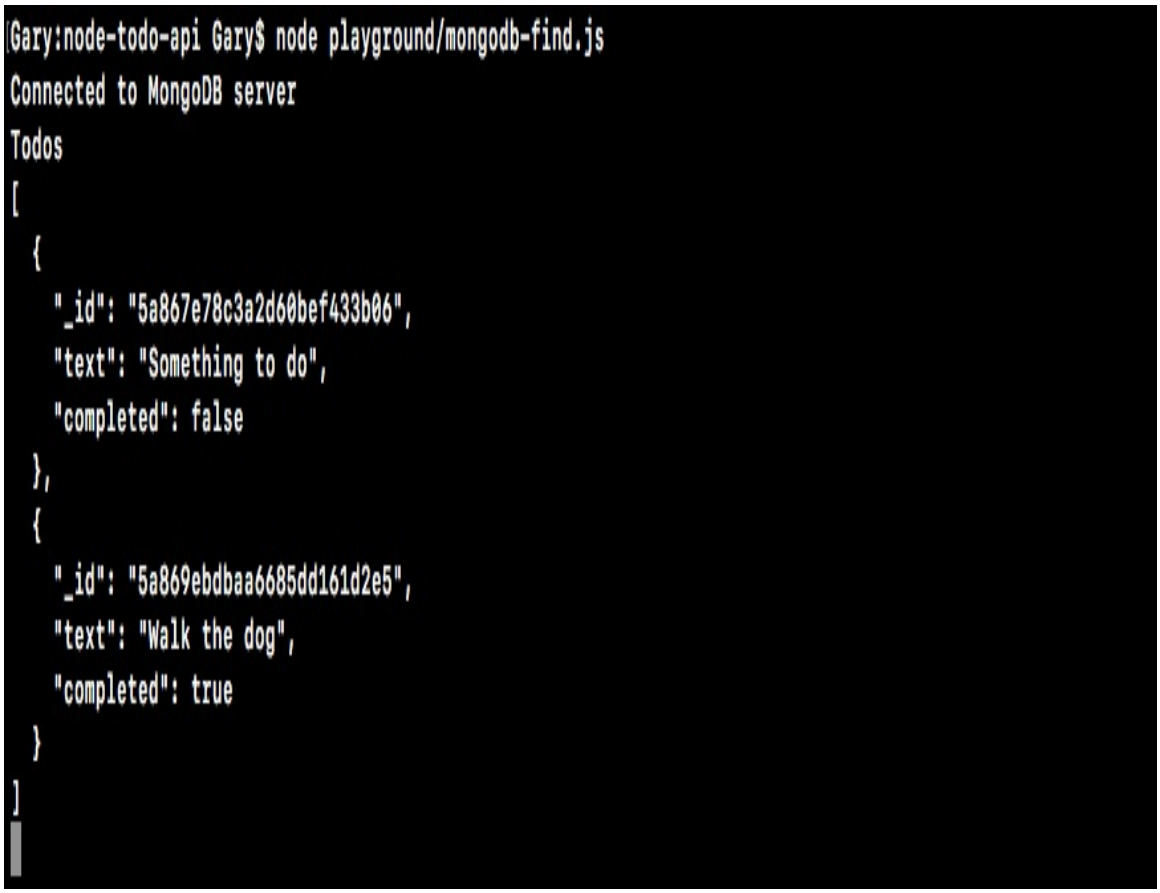


We have our `Todos` array with our two documents. We have our `_id`, our `text` properties, and our `completed` Boolean values. We now have a way to query our data right inside of Node.js. Now, this is a very basic query. We fetch everything in the `Todos` array, regardless of whether or not it has certain values.

Writing a query to fetch certain values

In order to query based on certain values, let's go ahead and switch up our `Todos`. Currently, both of them have a `completed` value equal to `false`. Let's go ahead and change the `Walk the dog` completed value to `true` so we can try to just query items that aren't completed. Over in Robomongo, I'm going to right-click the document and click Edit Document, and there we can edit the values. I'm going to change the `completed` value from `false` to `true`, and then I can save the record:

Inside of the Terminal, I can rerun the script to prove that it has changed. I'm going to shut down the script by running *control + C*, and then I can rerun it:



As shown in the preceding screenshot, we have our two `Todos`, one with a `completed` value of `false` and one with a `completed` value of `true`. By default, a Todo app is probably only going to show you the `Todos` collection you haven't completed. The ones you have completed, like `Walk the dog`, will probably be hidden, although they could be accessible if you clicked a button like Show all Todos. Let's go ahead and write a query that just fetches the `Todos` collection that have a `completed` status set to `false`.

Writing a query to fetch completed todos

To get this done, inside of Atom, we're going to make a change to how we call find. Instead of passing in `0` arguments, we're going to pass in `1`. This is what's known as our query. We can start specifying how we want to query the `Todos` collection. For example, maybe we want to query only `Todos` that have a `completed` value equal to `false`. All we have to do to query by value is set up the key-value pairs, as shown here:

```
db.collection('Todos').find({completed: false}).toArray().then((docs) => {
```

If I rerun our script over in the Terminal after shutting it down, we get just our one Todo item:

```
Gary:node-todo-api Gary$ node playground/mongodb-find.js
Connected to MongoDB server
Todos
[
  {
    "_id": "5a867e78c3a2d60bef433b06",
    "text": "Something to do",
    "completed": false
  }
]
```

We have our item with the `text` equal to `Something to do`. It has a `completed`

status of `false`, so it shows up. Our other Todo with a `text` property of `Walk the dog` is not showing up because that one has been completed. It doesn't match the query, so MongoDB does not return it. This is going to come in handy as we start querying our documents based off of completed values, text properties, or IDs. Let's take a quick moment to look at how we can query one of our `Todos` by ID.

Qureying todos by id

The first thing we need to do is remove everything from our query object; we no longer want to query by the `completed` value. Instead, we're going to query by the `_id` property.

Now, in order to illustrate this, I'm going to grab the ID of the Todo with the `completed` value of `false` from the Terminal. I'm going to copy it using *command + C*. If you're on Windows or Linux, you might need to right-click after highlighting the ID, and click Copy text. Now that I have the text inside of the clipboard, I can head over to the query itself. Now, if we try to add the ID like this:

```
db.collection('Todos').find({_id: ''}).toArray().then((docs) => {
```

It is not going to work as expected because what we have inside of the ID property is not a string. It's an ObjectId, which means that we need to use the `ObjectID` constructor function that we imported previously in order to create an ObjectId for the query.

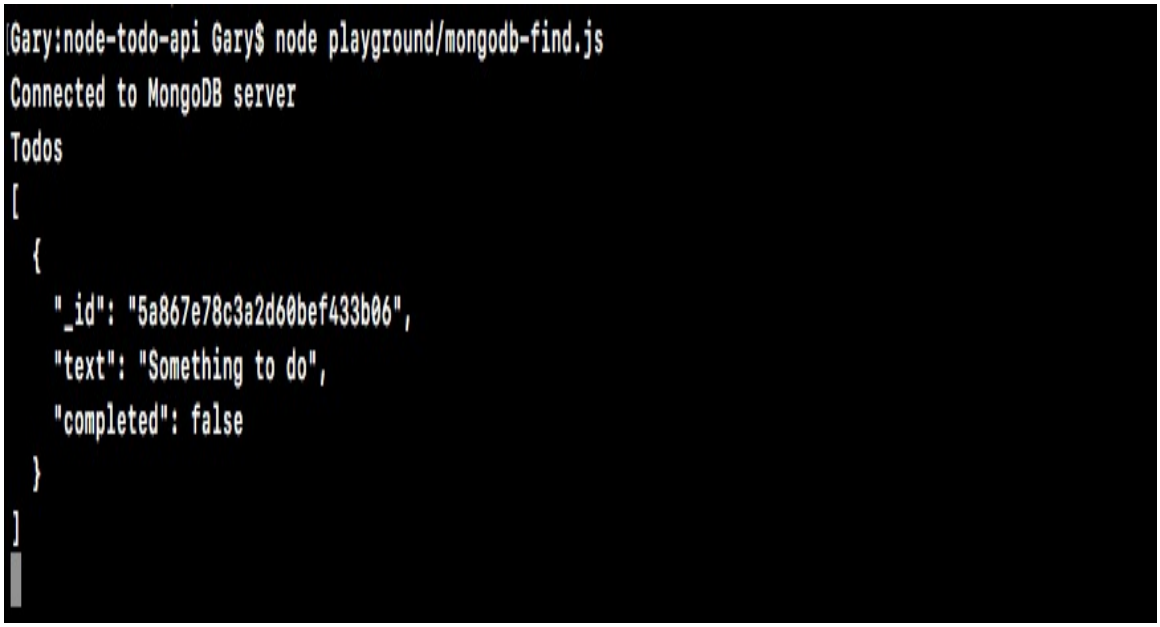
To illustrate how that's going to happen, I'm going to go ahead and indent our object. This is going to make it a little easier to read and edit.

```
db.collection('Todos').find({
  _id: '5a867e78c3a2d60bef433b06'
}).toArray().then((docs) => {
```

Now, I'm going to remove the string and call `new ObjectID`. The `new ObjectID` constructor does take an argument: the ID, in this case, we have it stored as a string. This is going to work as expected.

```
db.collection('Todos').find({
  _id: new ObjectID('5a867e78c3a2d60bef433b06');
})
```

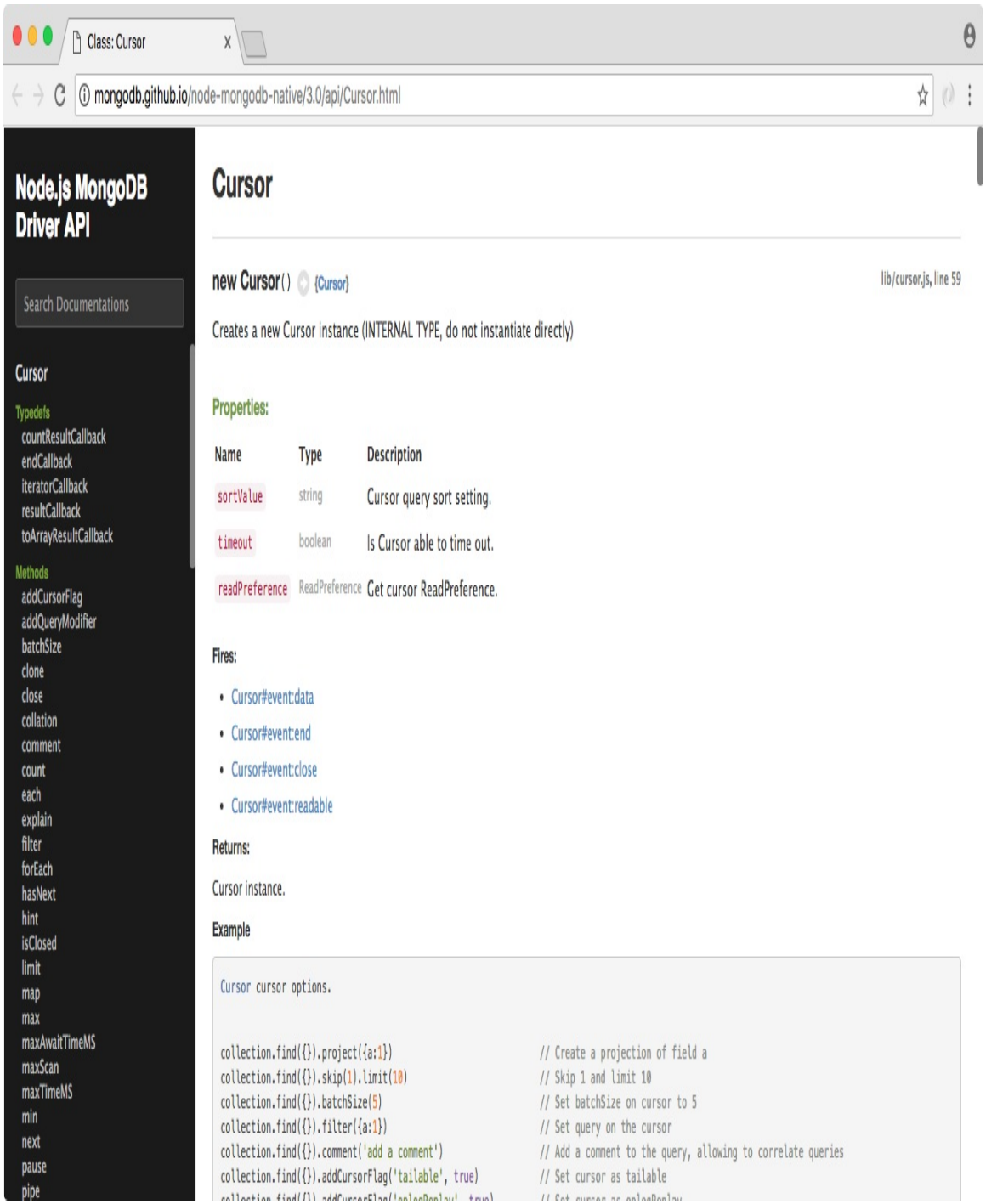
What we're doing here is we're querying the Todos collection, looking for any records that have an `_id` property equal to the ID we have. Now, I can go ahead and save this file, give things a refresh by running the script again, and we'll get the exact same Todo:



I can go ahead and change it for the `Walk the dog` Todo by copying the string value, pasting that inside of the ObjectID constructor function, and rerunning the script. When I do this, I get the `Walk the dog` Todo returned because that was the ObjectId I queried.

Now, querying in this fashion is one of the ways we'll be using find, but there are other methods other than `toArray` that are available on our cursors. We can explore other ones by heading over to the docs for the native driver. Inside of Chrome, have the MongoDB docs pulled up—these are the docs I showed you how to access in the previous chapter—and on the left-hand side, we have the Cursor section.

If you click that, we can view a list of all the methods available to us on that cursor:



This is what comes back from find. At the very bottom of the list, we have our `toArray` method. The one that we're going to look at right now is called count. From previous, you can go ahead and click

count; it's going to bring you to the documentation; the documentation for the native driver is actually really good. There is a complete list of all the arguments that you can provide. Some of them are optional, some of them are required, and there is usually a real-world example. Next, we can figure out exactly how to use `count`.

Implementing the count method

Now, we're going to go ahead and implement `count` over inside of Atom. What I'm going to do is take the current query, copy it to the clipboard, and then comment it out. I'm going to go ahead and replace our call to `toArray` with a call to `count`. Let's go ahead and remove the query that we pass in to find. What we're going to do here is count up all of the Todos in the `Todos` collection. Instead of having a call to `toArray`, we're going to have a call to count instead.

```
db.collection('Todos').find({}).count().then((count) => {
```

As you saw inside of the examples for count, they call count like this: calling count, passing in a callback function that gets called with an error, or the actual count. You can also have a promise as a way to access that data, which is exactly what we did with `toArray`. In our case, instead of passing a callback function like this, we're going to use the promise instead. We already have the promise set up. All we need to do to fix this is change `docs` to `count`, and then we're going to remove the `console.log` caller where we print the docs to the screen. Right after we print Todos, we're going to print `Todos count`, with a colon passing in the value.

```
db.collection('Todos').find({}).count().then((count) => {
   console.log('Todos count:');
}, (err) => {
   console.log('Unable to fetch todos', err);
});
```

This is not a template string, but I am going to go ahead and swap it

out with one, replacing the quotes with `. Now, I can pass in the `count`.

```
db.collection('Todos').find({}).count().then((count) => {
   console.log(`Todos count: ${count}`);
}, (err) => {
   console.log('Unable to fetch todos', err);
});
```

Now that we have this in place, we have a way to count up all of the Todos in the `Todos` collection. Inside the Terminal, I'm going to go ahead and shut down our previous script and rerun it:

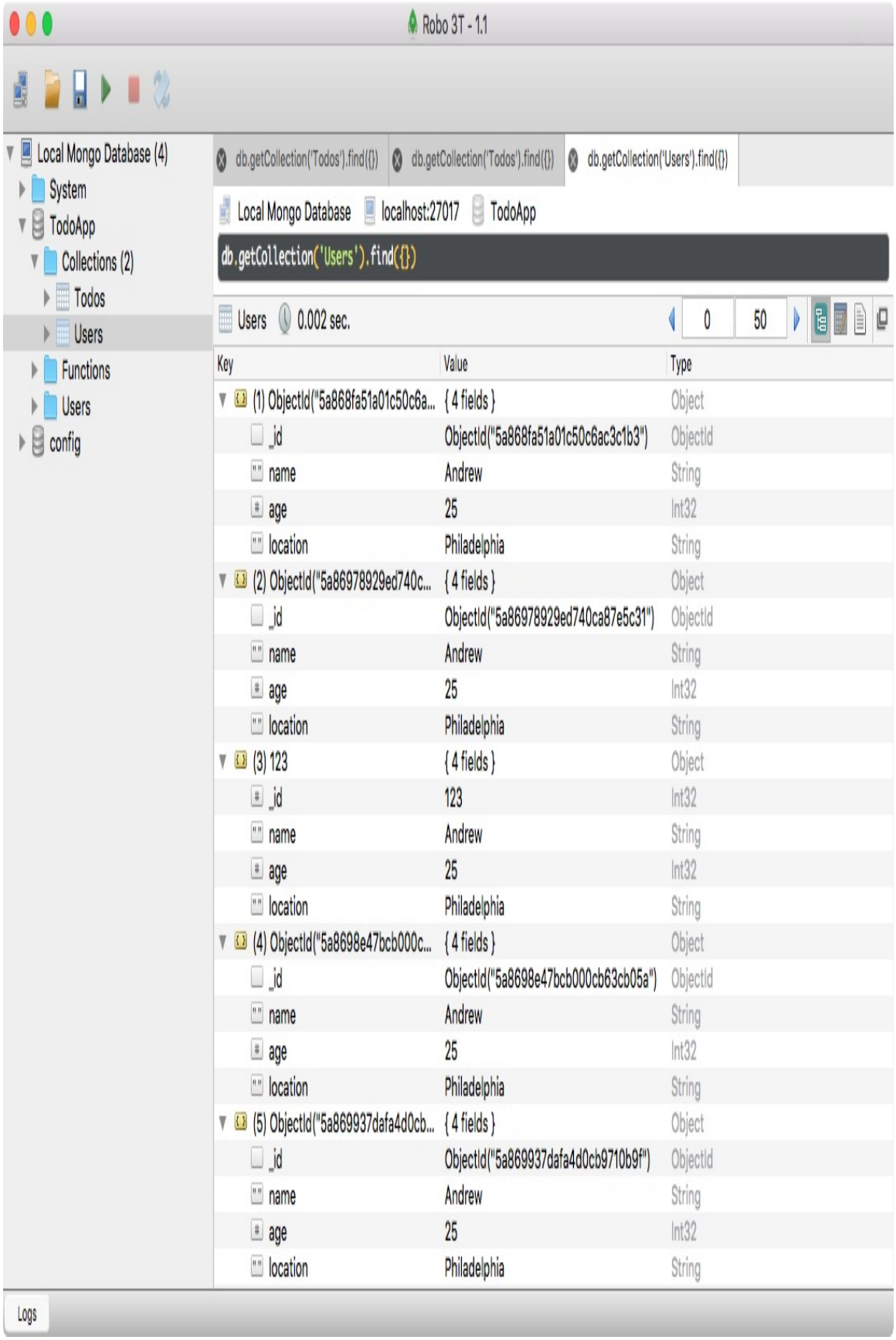
```
Gary:node-todo-api Gary$ node playground/mongodb-find.js
Connected to MongoDB server
Todos count: 2
```

We get `Todos count` too, which is correct. The cursor that we have, a call to find, returns everything in the Todos collection. If you count all of that up, you're going to get those two Todo items.

Once again, these are `count` and `toArray`; they're just a subset of all of the awesome methods you have available to you. We will be using other methods, whether it be the MongoDB native driver or, as you'll see later, the library Mongoose, but for now let's go ahead and do a challenge, given what you know.

Querying users collection

To get started, let's head into Robomongo, open up the Users collection, and take a look at all the documents we have inside of there. We currently have five. If you don't have the exact same number or yours are a little different, that's fine. I'm going to highlight them, right-click them, and click Expand Recursively. This is going to show me all of the key-value pairs for each document:


Robo 3T - 1.1
Local Mongo Database (4)
System
TodoApp
Collections (2)
Todos
Users
Functions
Users
config
db.getCollection('Todos').find({})
db.getCollection('Todos').find({})
db.getCollection('Users').find({})
Local Mongo Database
localhost:27017
TodoApp
db.getCollection('Users').find({})
Users
0.002 sec.
0
50
Key
Value
Type
(1) ObjectId("5a868fa51a01c50c6a... { 4 fields } Object
_id ObjectId("5a868fa51a01c50c6ac3c1b3") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
(2) ObjectId("5a86978929ed740c... { 4 fields } Object
_id ObjectId("5a86978929ed740ca87e5c31") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
(3) 123 { 4 fields } Object
_id 123 Int32
name Andrew String
age 25 Int32
location Philadelphia String
(4) ObjectId("5a8698e47bcb000c... { 4 fields } Object
_id ObjectId("5a8698e47bcb000cb63cb05a") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
(5) ObjectId("5a869937dafa4d0cb... { 4 fields } Object
_id ObjectId("5a869937dafa4d0cb9710b9f") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
Logs

Currently, aside from the ID, they're all identical. The name's Andrew, the age is 25, and the location is Philadelphia. I'm going to tweak the name property for two of them. I'm going to right-click the first document, and change the name to something like `Jen`. Then, I'll go ahead and do the same thing for the second document. I'm going to edit that document and change the name from `Andrew` to `Mike`. Now I have one document with a name of `Jen`, one with `Mike`, and three with `Andrew`.

We're going to query our users, looking for all of the users with the name equal to the name that you provided in the script. In this case, I'm going to try to query for all documents in the `Users` collection where the name is `Andrew`. Then, I'm going to print them into the screen, and I will expect to get three back. The two with the names `Jen` and `Mike` should not show up.

The first thing we need to do is fetch from the collection. This is going to be the `Users` collection as opposed to the `Todos` collection we've used in this chapter. In the `db.collection`, we're looking for the `Users` collection and now we're going to go ahead and call `find`, passing in our query. We want a query, fetching all documents where the `name` is equal to the string `Andrew`.

```
db.collection('Users').find({name: 'Andrew'})
```

This is going to return the cursor. In order to actually get the documents, we have to call `toArray`. We now have a promise; we can attach a `then` call onto `toArray` to do something with the `docs`. The documents are going to come back as the first argument in our success handler, and right inside of the function itself we can print the docs to the screen. I'm going to go ahead and use `console.log(JSON.stringify())`, passing in our three classic arguments: the object itself, `docs`, `undefined`, and `2` for formatting:

```
db.collection('Users').find({name: 'Andrew'}).toArray().then((docs) => {
```

```
  console.log(JSON.stringify(docs, undefined, 2));
});
```

With this in place, we have now done. We have a query, and it should work. We can test it by running it from the Terminal. Inside the Terminal, I'm going to go ahead and shut down the previous connection and rerun the script:

```
Gary:node-todo-api Gary$ node playground/mongodb-find.js
Connected to MongoDB server
[
  {
    "_id": 123,
    "name": "Andrew",
    "age": 25,
    "location": "Philadelphia"
  },
  {
    "_id": "5a8698e47bcb000cb63cb05a",
    "name": "Andrew",
    "age": 25,
    "location": "Philadelphia"
  },
  {
    "_id": "5a869937dafa4d0cb9710b9f",
    "name": "Andrew",
    "age": 25,
    "location": "Philadelphia"
  }
]
```

When I do this, I get my three documents back. All of them have a `name` equal to `Andrew`, which is correct because of the query we set up.

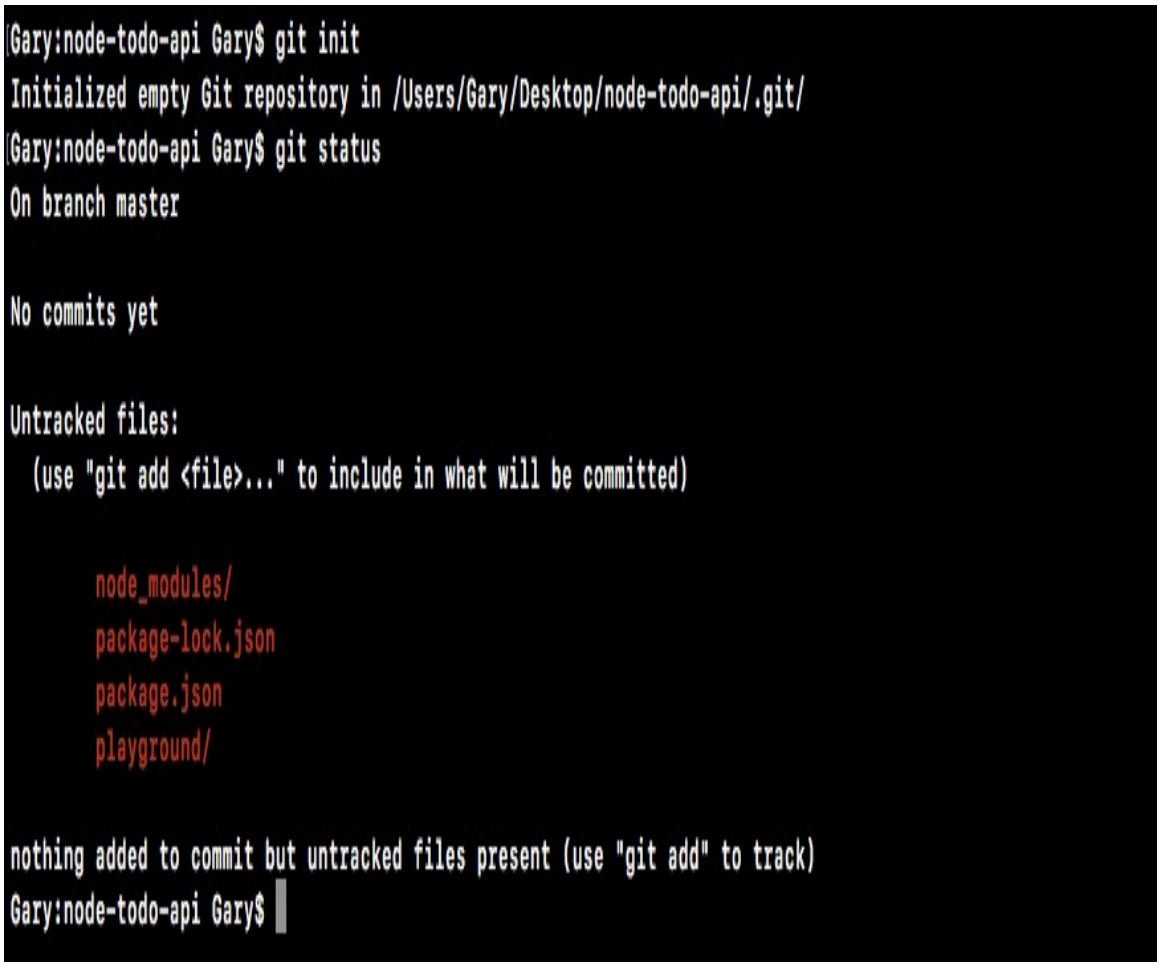
Notice the documents with a name equal to `Mike` or `Jen` are nowhere to be found.

We now know how to insert and query data from the database. Up next, we're going to take a look at how we can remove and update documents.

Setting up the repo

Before we go any further, I do want to add version control to this project. In this section, we're going to create a new repo locally, make a new GitHub repository, and push our code to that GitHub repository. If you're already familiar with Git or GitHub, you can go ahead and do that on your own; you don't need to go through this section. If you're new to Git and it doesn't make sense just yet, that's also fine. Simply follow along, and we'll go through the whole process.

This section is going to be really simple; nothing MongoDB- related here. To get started, I am going to go ahead and initialize a new Git repository from the Terminal by using `git init`. This is going to initialize a new repository, and I can always run `git status` like this to take a look at the files that are untracked:



Here we have our `playground` folder, which we want to add under version control, and we have `package.json`. We also have `node_modules`. We do not want to track this directory. This contains all of our npm libraries. To ignore `node_modules`, in Atom we're going to make the `.gitignore` file in the root of our project. If you remember, this lets you specify files and folders that you want to leave out of your version control. I'm going to create a new file called `.gitignore`. In order to ignore the `node_modules` directory, all we have to do is type it exactly as it's shown here:

```
node_modules/
```

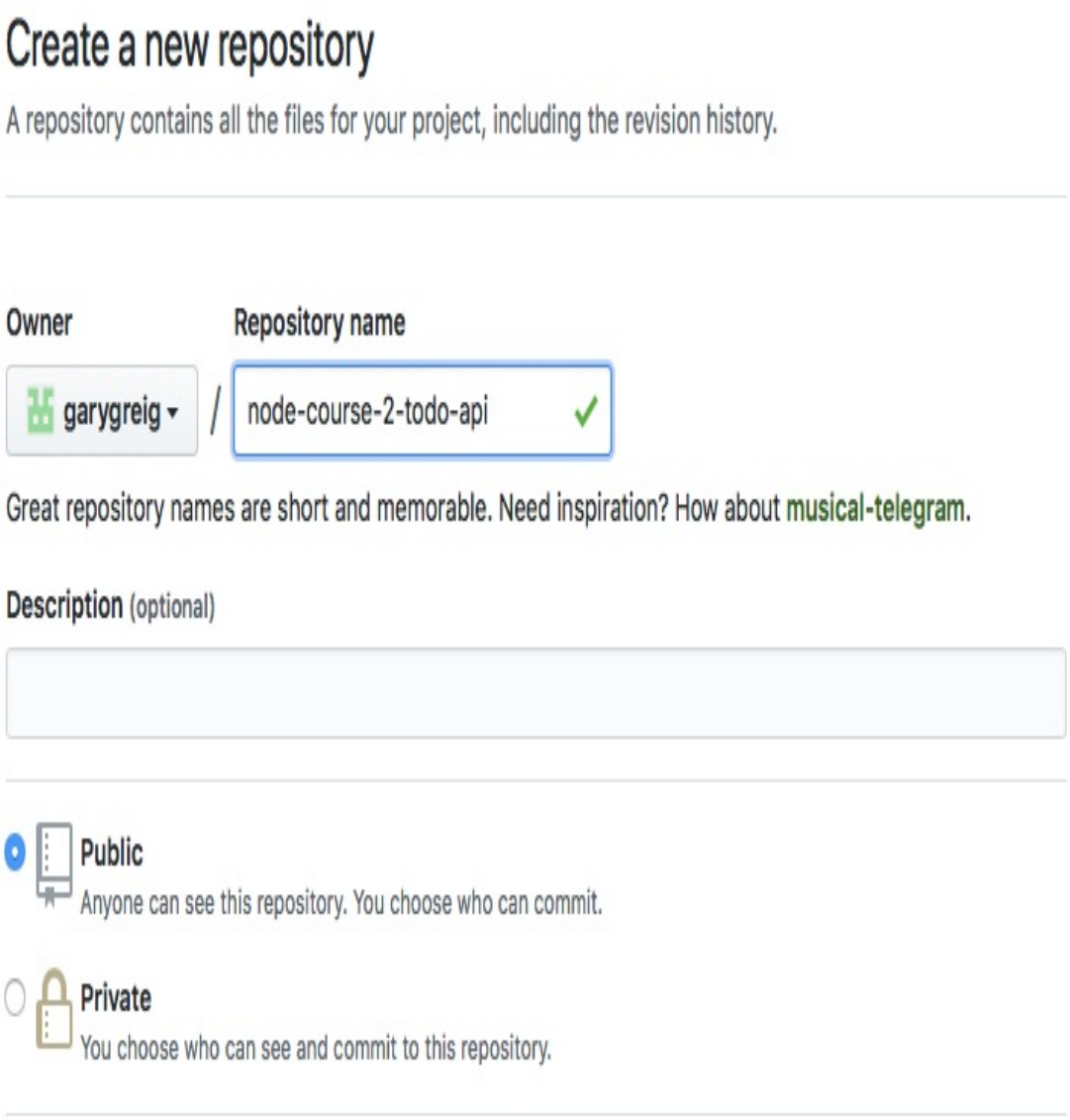
I'm going to save the file and rerun `git status` from the Terminal. We get the `.gitignore` folder showing up, and the `node_modules` folder is

nowhere in sight:

The next thing we're going to do is make our first commit, using two commands. First up, I'm going to use `git add .` to add everything to the next commit. Then, I can make the commit using `git commit` with the `-m` flag. A good message for this commit would be `Init commit`:

```
git add .
git commit -m 'Init commit'
```

Now before we go, I do want to make a GitHub repository and get this code up there. This is going to require me to open up the browser and go to `github.com`. Once you're logged in we can make a new repo. I'm going to make a new repo and give it a name:



I'm going to go with `node-course-2-todo-api`. You can name yours something else if you wish. I'm going to go with this one to keep the course files organized. Now I can go ahead and create this repository, and as you may recall, GitHub actually gives us a few helpful commands:

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/garygreig/node-course-2-todo-api.git

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# node-course-2-todo-api" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/garygreig/node-course-2-todo-api.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/garygreig/node-course-2-todo-api.git
git push -u origin master
```

...or import code from another repository

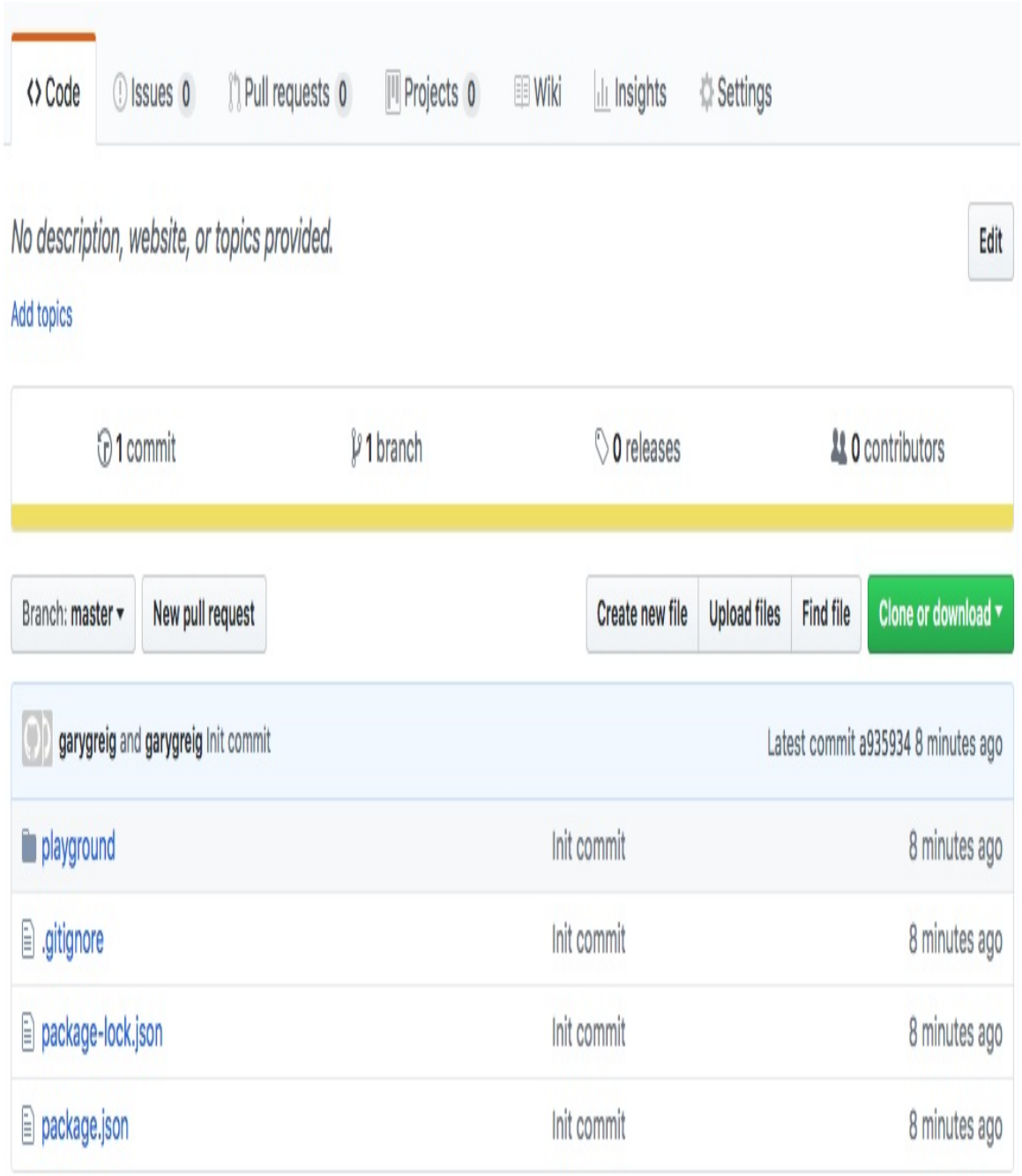
You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

Import code

In this case, we're pushing an existing repository from the command line. We already went through the steps of initializing the repository, adding our files and making our first commit. That means I can take the following two lines, copy them, head over to the Terminal, and paste them in:

```
git remote add origin https://github.com/garygreig/node-course-2-todo-api.git
git push -u origin master
```

You might need to do these one at a time, depending on your operating system. On the Mac, when I try to paste in multiple commands it's going to run all but the last, and then I just have to hit enter to run the last one. Take a moment to knock that out for your operating system. You might need to run it as one command, or you might be able to paste it all in and hit *enter*. Either way, what we have here is our code pushed up to GitHub. I can prove that it's pushed up by refreshing the repository page:



Right there we have all of our source code, the `.gitignore` file, `package.json`, and we have our `playground` directory with our MongoDB scripts.

That's it for this section. We'll explore how to delete data from a MongoDB collection in the next section.

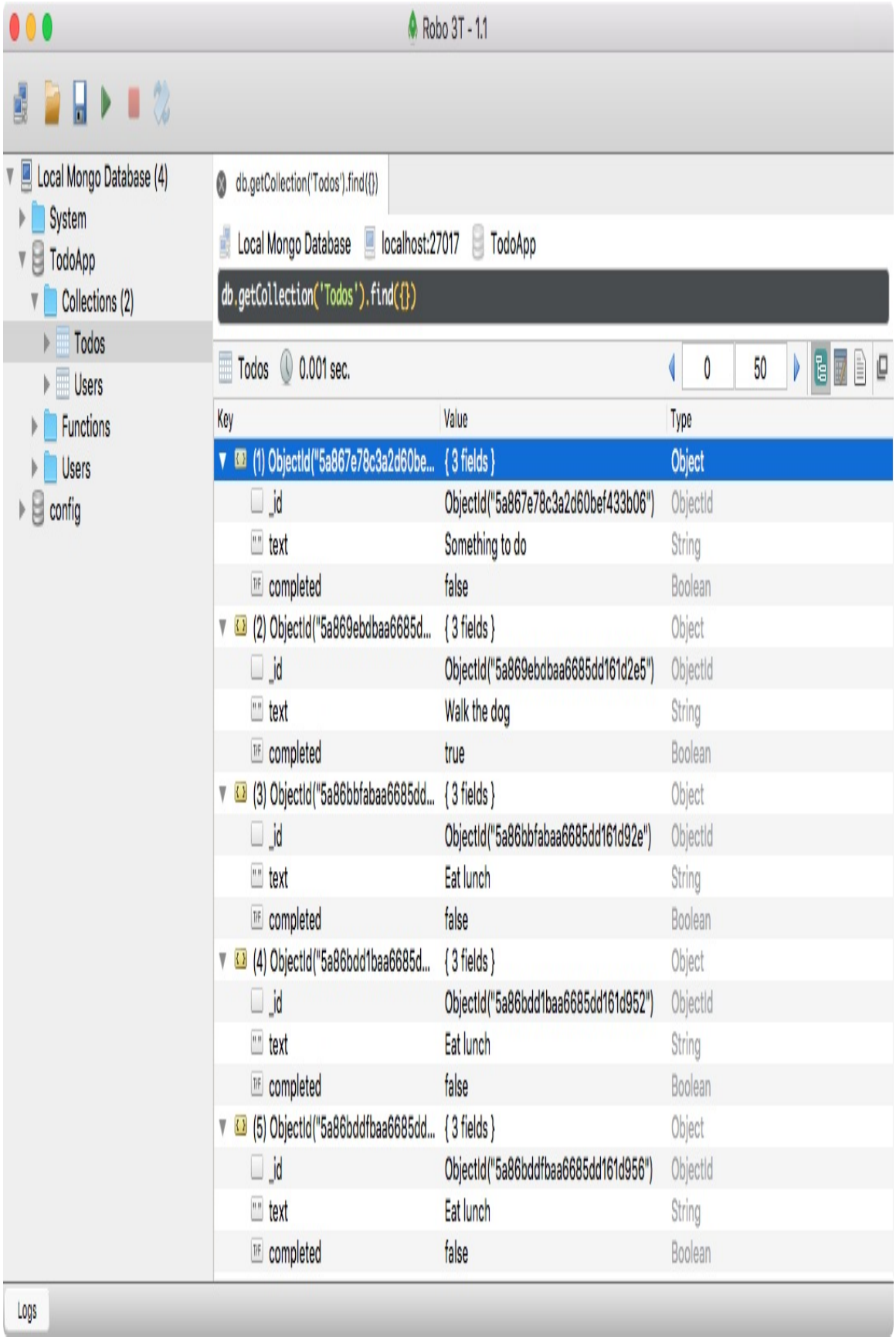
Deleting documents

In this section, you're going to learn how to delete documents from your MongoDB collections. Before we get into that, in order to explore the methods that let us delete multiple documents or just one, we want to create a few more Todos. Currently, the `Todos` collection only has two items, and we're going to need a few more in order to play around with all these methods involving deletion.

Now, I do have two. I'm going to go ahead and create a third by right-clicking and then going to Insert Document.... We'll make a new document with a `text` property equal to something like `Eat lunch`, and we'll set `completed` equal to `false`:

```
{
   text: 'Eat lunch',
   completed: false
}
```

Now before we save this, I am going to copy it to the clipboard. We're going to create a few duplicate Todos so we can see how we can delete items based off of specific criteria. In this case, we're going to be deleting multiple Todos with the same text value. I'm going to copy that to the clipboard, click Save, and then I'll create two more with the exact same structure. Now we have three Todos that are identical except for the ID, and we have two that have unique text properties:


Robo 3T - 1.1
Local Mongo Database (4)
System
TodoApp
Collections (2)
Todos
Users
Functions
Users
config
db.getCollection('Todos').find({})
Local Mongo Database
localhost:27017
TodoApp
db.getCollection('Todos').find({})
Todos
0.001 sec.
0
50
Key
Value
Type
(1) ObjectId("5a867e78c3a2d60be... { 3 fields } Object
_id ObjectId("5a867e78c3a2d60bef433b06") ObjectId
text Something to do String
completed false Boolean
(2) ObjectId("5a869ebdbaa6685d... { 3 fields } Object
_id ObjectId("5a869ebdbaa6685dd161d2e5") ObjectId
text Walk the dog String
completed true Boolean
(3) ObjectId("5a86bbfabaa6685dd... { 3 fields } Object
_id ObjectId("5a86bbfabaa6685dd161d92e") ObjectId
text Eat lunch String
completed false Boolean
(4) ObjectId("5a86bdd1baa6685d... { 3 fields } Object
_id ObjectId("5a86bdd1baa6685dd161d952") ObjectId
text Eat lunch String
completed false Boolean
(5) ObjectId("5a86bddfbaa6685dd... { 3 fields } Object
_id ObjectId("5a86bddfbaa6685dd161d956") ObjectId
text Eat lunch String
completed false Boolean
Logs

Let's go ahead and move into Atom and start writing some code.

Exploring methods to delete data

I'm going to duplicate the `mongodb-find` file, creating a brand-new file called `mongodb-delete.js`. In here, we'll explore the methods for deleting data. I'm also going to remove all of the queries that we set up in the previous section. I am going to keep the `db.close` method commented out, as once again we don't want to close the connection just yet; it's going to interfere with these statements we're about to write.

Now, there are three methods that we'll be using in order to remove data.

- The first one is going to be `deleteMany`. The `deleteMany` method will let us target many documents and remove them.
- We'll also be using `deleteOne`, which targets one document and removes it.
- And finally, we'll be using `findOneAndDelete`. The `findOneAndDelete` method lets you remove an individual item and it also returns those values. Imagine I want to delete a Todo. I delete the Todo, but I also get the Todo object back so I can tell the user exactly which one got deleted. This is a really useful method.

The deleteMany method

Now, we're going to start off with `deleteMany`, and we're going to target those duplicates we just created. The goal in this section, is to delete every single Todo inside of the Todos collection that has a `text` property equal to `Eat lunch`. Currently, there are three out of five that fit that criteria.

In Atom, we can go ahead and kick things off by doing `db.collection`. This is going to let us target our Todos collection. Now, we can go ahead and use the collection method `deleteMany`, passing in the arguments. In this case, the only argument we need is our object, and this object is just like the object we passed to find. With this, we can target our Todos. In this case, we're going to delete every Todo where the `text` equals `Eat lunch`.

```
//deleteMany
db.collection('Todos').deleteMany({text: 'Eat lunch'});
```

We didn't use any punctuation in RoboMongo, so we're also going to avoid punctuation over in Atom; it needs to be exactly the same.

Now that we have this in place, we could go ahead and tack on a `then` call to do something when it either succeeds or fails. For now, we'll just add a success case. We are going to get a result argument passed back to the callback, and we can print that to the `console.log(result)` screen, and we'll take a look at exactly what is in this result object a bit later.

```
//deleteMany
db.collection('Todos').deleteMany({text: 'Eat lunch'}).then((result) => {
  console.log(result);
```

```
});
```

With this in place, we now have a script that deletes all Todos where the text value is `Eat lunch`. Let's go ahead and run it, and see exactly what happens. In the Terminal, I'm going to run this file. It's in the `playground` folder, and we just called it `mongodb-delete.js`:

```
node playground/mongodb-delete.js
```

Now when I run it, we get a lot of output:

```
Gary:node-todo-api Gary$ node playground/mongodb-delete.js
Connected to MongoDB server
CommandResult {
  result: { n: 3, ok: 1 },
  connection:
   Connection {
     domain: null,
     _events:
      { error: [Function],
        close: [Function],
        timeout: [Function],
        parseError: [Function] },
     _eventsCount: 4,
     _maxListeners: undefined,
     options:
      { host: 'localhost',
        port: 27017,
        size: 5,
```

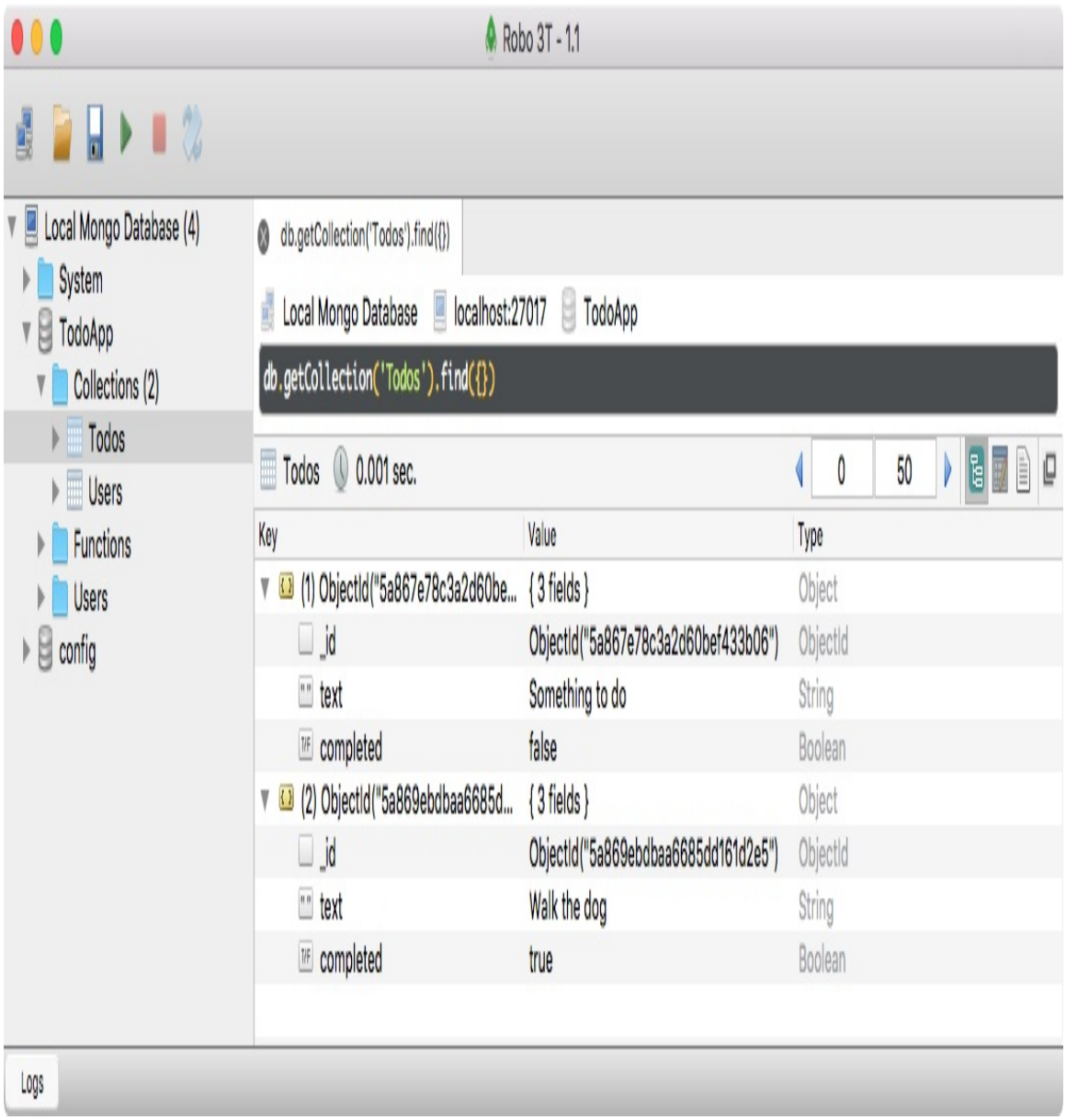
A really important piece of output, the only important piece actually, is up at the very top. If you scroll to the top, what you're

going to see is this `result` object. We get `ok` set to `1`, which means things did go as expected, and we get `n` set to `3`. `n` is the number of records that were deleted. In this case, we had three Todos that match that criteria, so three Todos were deleted. This is how you can target and delete many Todos.

The deleteOne Method

Now, aside from `deleteMany`, we have `deleteOne`, and `deleteOne` works exactly the same as `deleteMany`, only it deletes the first item it sees that matches the criteria and then it stops.

To illustrate exactly how this works, we're going to create two items inside of our collection. If I give things a refresh, you will see that we now only have two documents:



These are the ones we started with. I'm going to insert documents again using the same data that's already in my clipboard. This time we'll just make two document, two that are identical.

The deleteOne method

The goal here is to use `deleteOne` to delete the document where the text equals `Eat lunch`, but since we're using `deleteOne` and not `deleteMany`, one of these should stay around and one of them should go away.

Back inside of Atom, we can go ahead and get started by calling `db.collection` with the collection name we want to target. In this case it's `Todos` again, and we're going to use `deleteOne`. The `deleteOne` method takes that same criteria. We're going to target documents where `text` equals `Eat lunch`.

This time though, instead of deleting multiple documents we're just going to delete the one, and we are still going to get that same exact result. To prove it, I'll just print to the screen like we did previously with `console.log(result)`:

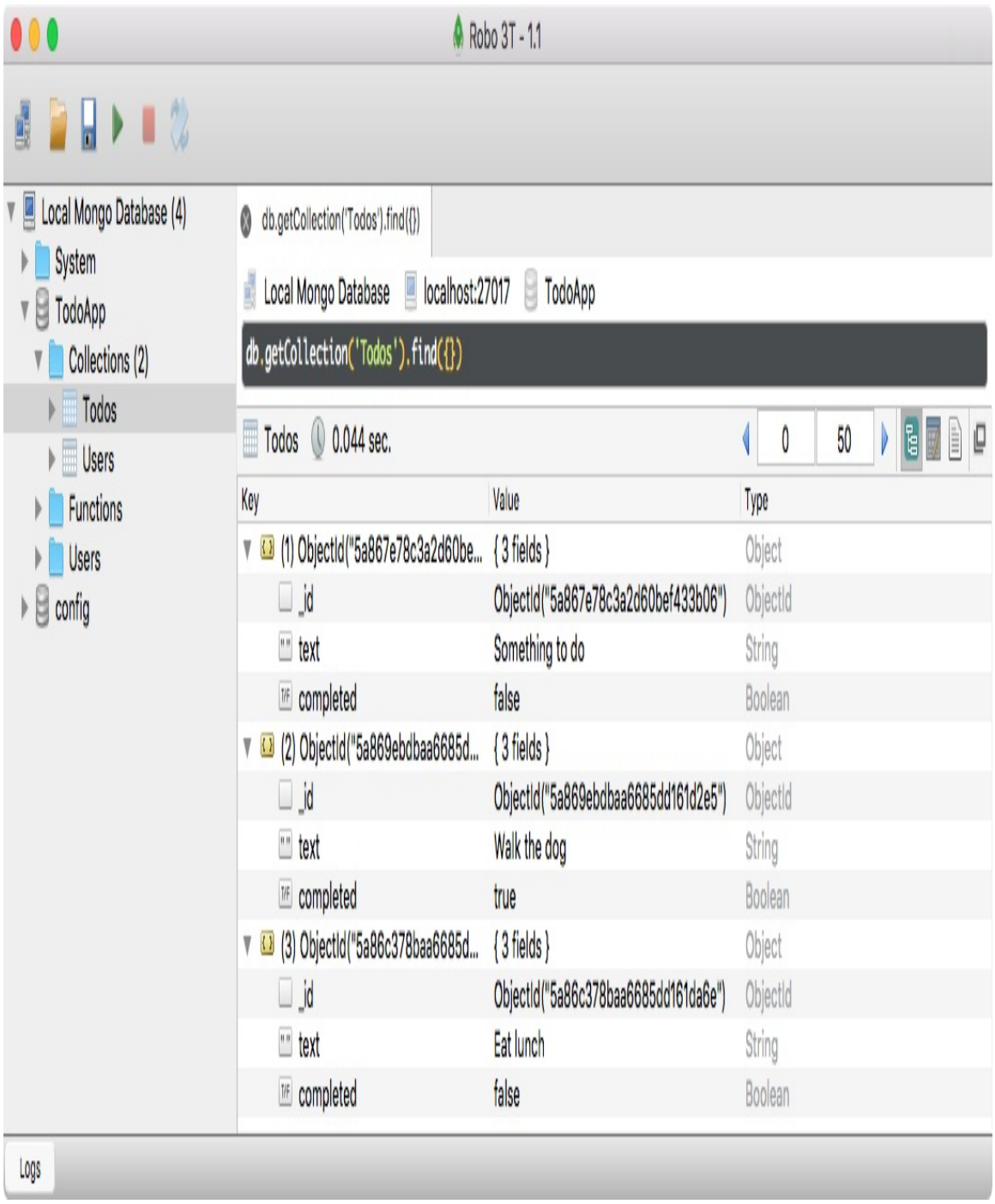
```
//deleteOne
db.collection('Todos').deleteOne({text: 'Eat lunch'}).then((result) => {
  console.log(result);
});
```

With this in place, we can now rerun our script and see what happens. In the Terminal, I'm going to shut down our current connection and rerun it:

```
Gary:node-todo-api Gary$ node playground/mongodb-delete.js
Connected to MongoDB server
CommandResult {
  result: { n: 1, ok: 1 },
  connection:
   Connection {
     domain: null,
     _events:
      { error: [Function],
        close: [Function],
        timeout: [Function],
        parseError: [Function] },
     _eventsCount: 4,
     _maxListeners: undefined,
     options:
      { host: 'localhost',
        port: 27017,
        size: 5,
```

We get a similar-looking object, a bunch of junk we don't really care about, but once again if we scroll to the top we have a `result` object, where `ok` is `1` and the number of deleted documents is also `1`. Even though multiple documents did pass this criteria it only deleted the first one, and we can prove that by going over to Robomongo, right-clicking up above, and viewing the documents again. This time around, we have three Todos.

We do still have one of the Todos with the `Eat lunch` text:



And now that we know how to use these two methods, I want to take a look at my favorite method. This is `findOneAndDelete`.

The findOneAndDelete method

Most of the time, when I'm deleting a document, I only have the ID. This means that I don't exactly know what the text is or the completed status, and that can be really useful depending on your user interface. For example, if I delete a Todo, maybe I want to show that next, saying *You deleted the Todo that says Eat lunch*, with a little undo button in case they didn't mean to take that action. Getting the data back as well as deleting it can be really useful.

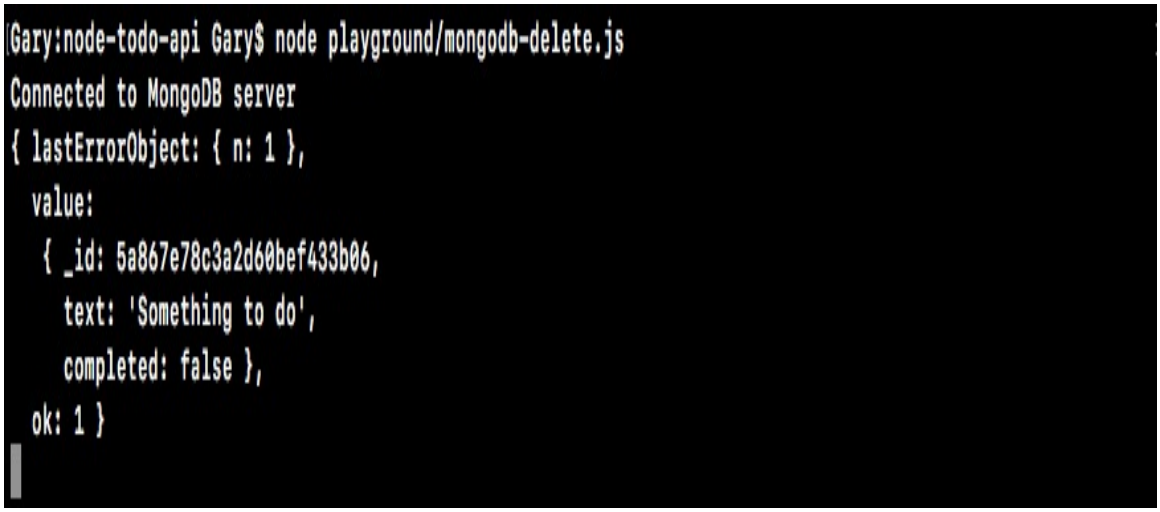
In order to explore `findOneAndDelete`, we're going to once again target the Todo where the `text` equals `Eat lunch`. I'm going to go ahead and comment out `deleteOne`, and next we can get started by accessing the appropriate collection. The method is called `findOneAndDelete`. The `findOneAndDelete` method takes a very similar set of arguments. The only thing we need to pass in is the query. This is going to be identical to the ones we have in the previous screenshot. This time though, let's go ahead and target Todos that had a `completed` value set to `false`.

Now there are two Todos that fit this query, but once again we're using a `findOne` method, which means it's only going to target the first one it sees, the one with a `text` property of `Something to do`. Back in Atom, we can get this done by targeting Todos where `completed` equals `false`. Now, instead of getting back a result object with an `ok` property and an `n` property, the `findOneAndDelete` method actually gets that document back. This means we can tack on a `then` call, we can get our result, and we can print it to the screen once again with `console.log(result)`:

```
//findOneAndDelete
db.collection('Todos').findOneAndDelete({completed: false}).then((result) => {
```

```
  console.log(result);
});
```

Now that we have this in place, let's test things out over in the Terminal. In the Terminal, I'm going to shut down the script and start it up again:



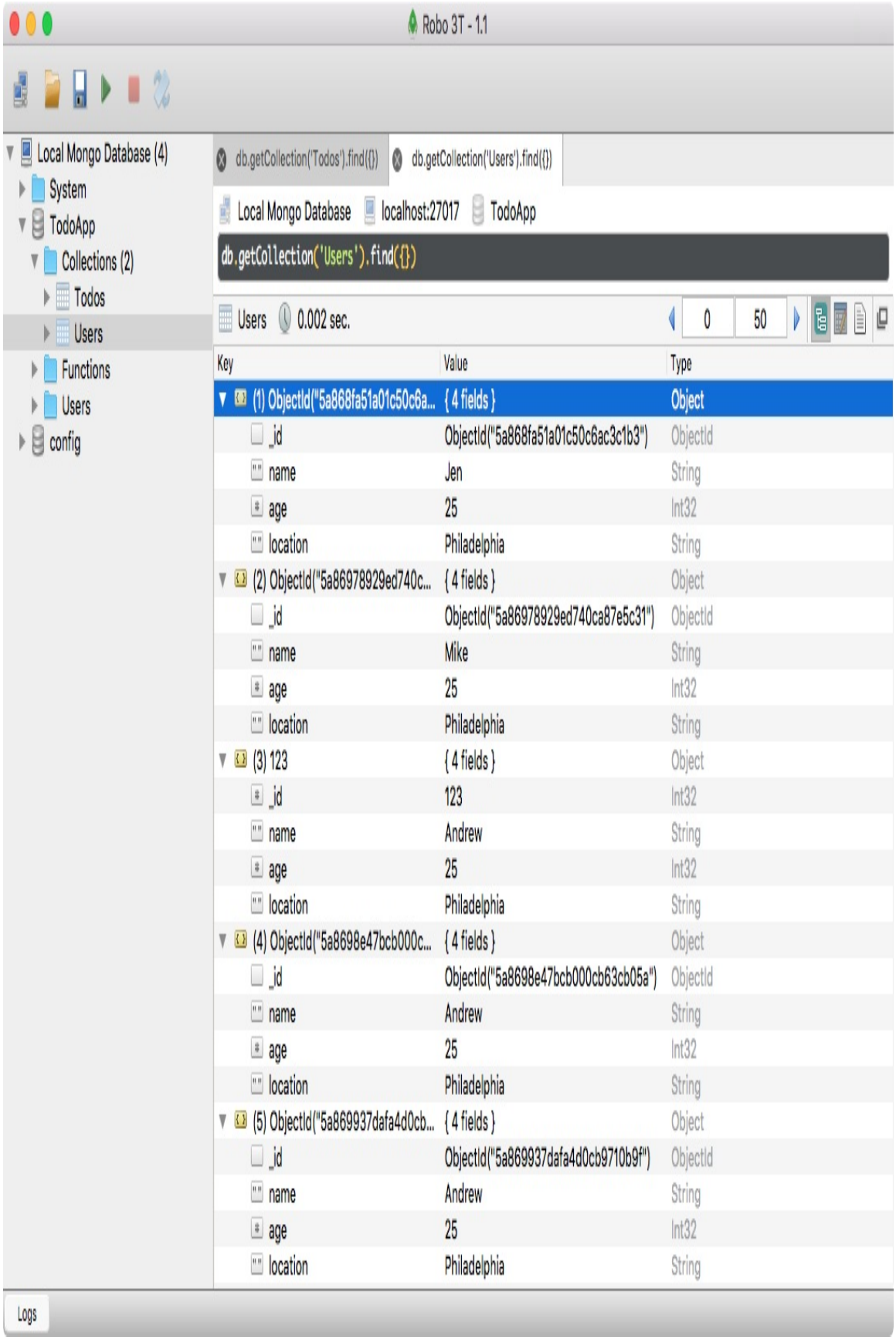
We get a few different things in our result object. We do get an `ok` set to `1`, letting us know things went as planned. We have a `lastErrorObject`; we'll talk about that in just a second; and we have our `value` object. This is the actual document we deleted. This is why the `findOneAndDelete` method is super handy. It gets that document back as well as deleting it.

Now in this particular case, the `lastErrorObject`, once again just has our `n` property, and we can see the number of Todos that were deleted. There is other information that could potentially be in `lastErrorObject`, but that's only going to happen when we use other methods, so we'll look at that when the time comes. For now, when you delete a Todo, we just get the number back.

With this in place, we now have three different ways we can target our MongoDB documents and remove them.

Using the deleteMany and findOneAndDelete methods

We're going to go ahead and go over a quick challenge to test your skills. Inside of Robomongo, we can look at the data we have in the `Users` collection. I'm going to open it up, highlight all the data, and expand it recursively so we can view it:


Robo 3T - 1.1
Local Mongo Database (4)
System
TodoApp
Collections (2)
Todos
Users
Functions
Users
config
db.getCollection('Todos').find({})
db.getCollection('Users').find({})
Local Mongo Database
localhost:27017
TodoApp
db.getCollection('Users').find({})
Users
0.002 sec.
0
50
Key
Value
Type
(1) ObjectId("5a868fa51a01c50c6a... { 4 fields } Object
_id ObjectId("5a868fa51a01c50c6ac3c1b3") ObjectId
name Jen String
age 25 Int32
location Philadelphia String
(2) ObjectId("5a86978929ed740c... { 4 fields } Object
_id ObjectId("5a86978929ed740ca87e5c31") ObjectId
name Mike String
age 25 Int32
location Philadelphia String
(3) 123 { 4 fields } Object
_id 123 Int32
name Andrew String
age 25 Int32
location Philadelphia String
(4) ObjectId("5a8698e47bcb000c... { 4 fields } Object
_id ObjectId("5a8698e47bcb000cb63cb05a") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
(5) ObjectId("5a869937dafa4d0cb... { 4 fields } Object
_id ObjectId("5a869937dafa4d0cb9710b9f") ObjectId
name Andrew String
age 25 Int32
location Philadelphia String
Logs

We have the name Jen; we have Mike; we have Andrew, Andrew and Andrew. This is perfect data. Yours might look a little different, but the goal is to use two methods. First up, look for any duplicates, anything that has a name set to the name of another document. In this case, I have three documents where the name is Andrew. What I want to do is use `deleteMany` to target all of these documents and remove them. I also want to use `findOneAndDelete` to delete another document; it doesn't matter which one. And I want you to delete it by ID.

In the end, both statements should show their effect over inside of Robomongo. When I'm done, I'm hoping to see these three documents deleted. They all have the name Andrew, and I'm hoping to see the document where the name Mike is deleted, because I'm going to target this one by ID in my `findOneAndDelete` method call.

First up, I'm going to write my scripts, one for deleting users where the name is `Andrew` and one for deleting the document with the ID. In order to grab the ID, I am going to go ahead and edit it and simply grab the text inside of quotes, and then I can cancel the update and move into Atom.

Removing duplicate documents

First up, we're going to go ahead and try to remove the duplicate users, and I'm going to do this by using `db.collection`. We're going to target the `Users` collection, and in this particular case, we're going to be using the `deleteMany` method. Here, we're going to try to delete all of the users where the `name` property equals `Andrew`.

```
db.collection('Users').deleteMany({name: 'Andrew'});
```

Now I could tack on a then call to check for success or errors, or I could just leave it like this, which is what I'm going to do. If you use a callback or the promise then method, that is perfectly fine. As long as the deletion happens, you're good to go.

Targeting the documents using ID

Next up, I'm going to write the other statement. We're going to target the `Users` collection once again. Now, we're going to go ahead and use the `findOneAndDelete` method. In this particular case, I am going to be deleting the Todo where the `_id` equals the ObjectId I have copied to the clipboard, which means I need to create a `new ObjectID`, and I also need to go ahead and pass in the value from the clipboard inside of quotes.

```
db.collection('Users').deleteMany({name: 'Andrew'});

db.collection('Users').findOneAndDelete({
  _id: new ObjectID("5a86978929ed740ca87e5c31")
})
```

Either single or double would work. Make sure the capitalization of `ObjectID` is identical to what you have defined, otherwise this creation will not happen.

Now that we have the ID created and passed in as the `_id` property, we can go ahead and tack on a `then` callback. Since I'm using `findOneAndDelete`, I am going to print that document to the screen. Right here I'll get my argument, `results`, and I'm going to print it to the screen using our pretty- printing method,`console.log(JSON.stringify())`, passing in those three arguments, the `results`, `undefined`, and the spacing, which I'm going to use as `2`.

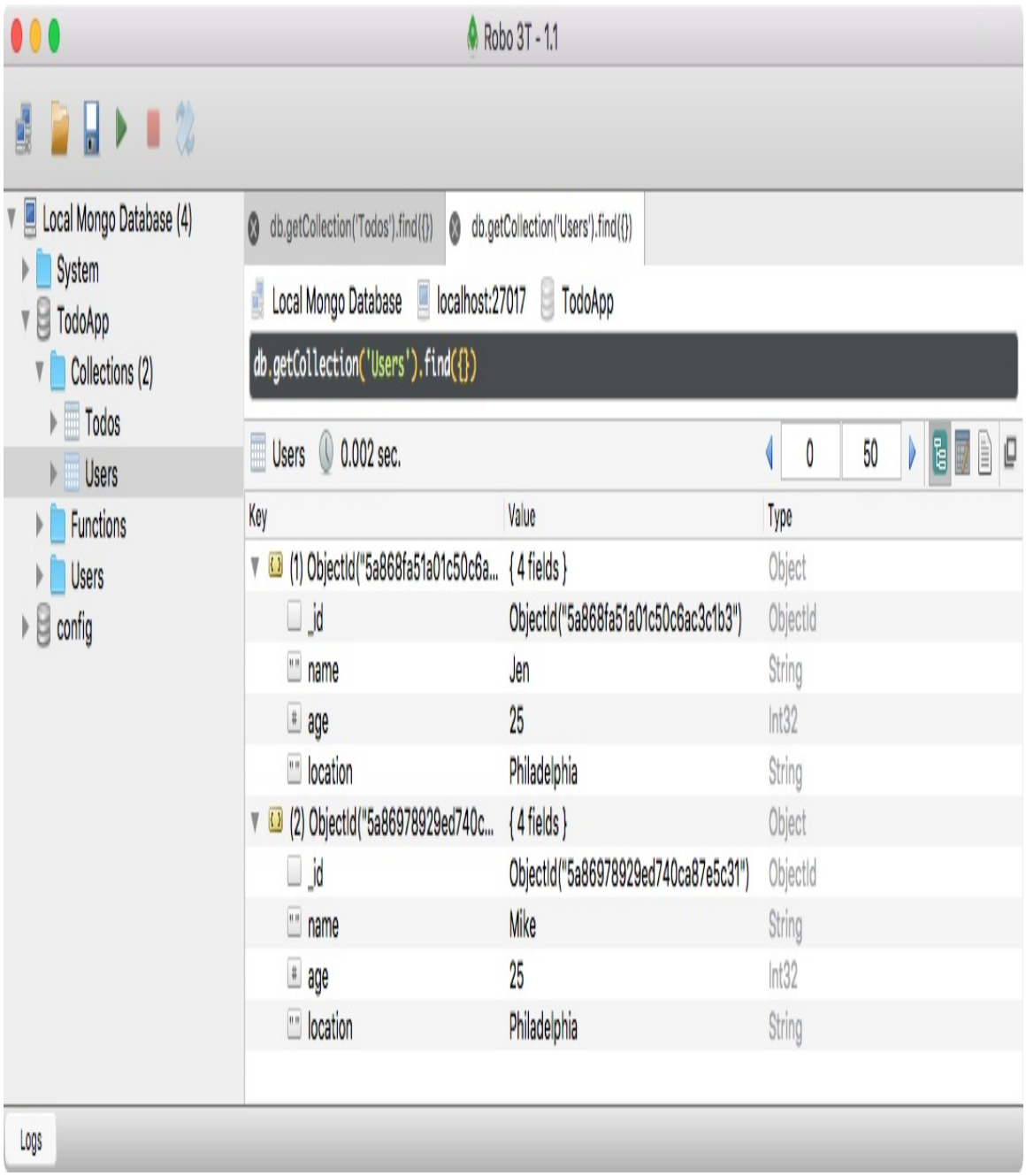
```
db.collection('Users').deleteMany({name: 'Andrew'});

db.collection('Users').findOneAndDelete({
```

```
  _id: new ObjectID("5a86978929ed740ca87e5c31")
}).then((results) => {
  console.log(JSON.stringify(results, undefined, 2));
});
```

With this in place, we are now ready to go.

Running the findOneAndDelete and deleteMany statements

Let's go ahead and comment out `findOneAndDelete` first. We'll run the `deleteMany` statement. Over in the Terminal, I can shut down the current connection, start it up again, and if we go over to Robomongo, we should see that those three documents were deleted. I'm going to right-click on `Users` and view the documents:



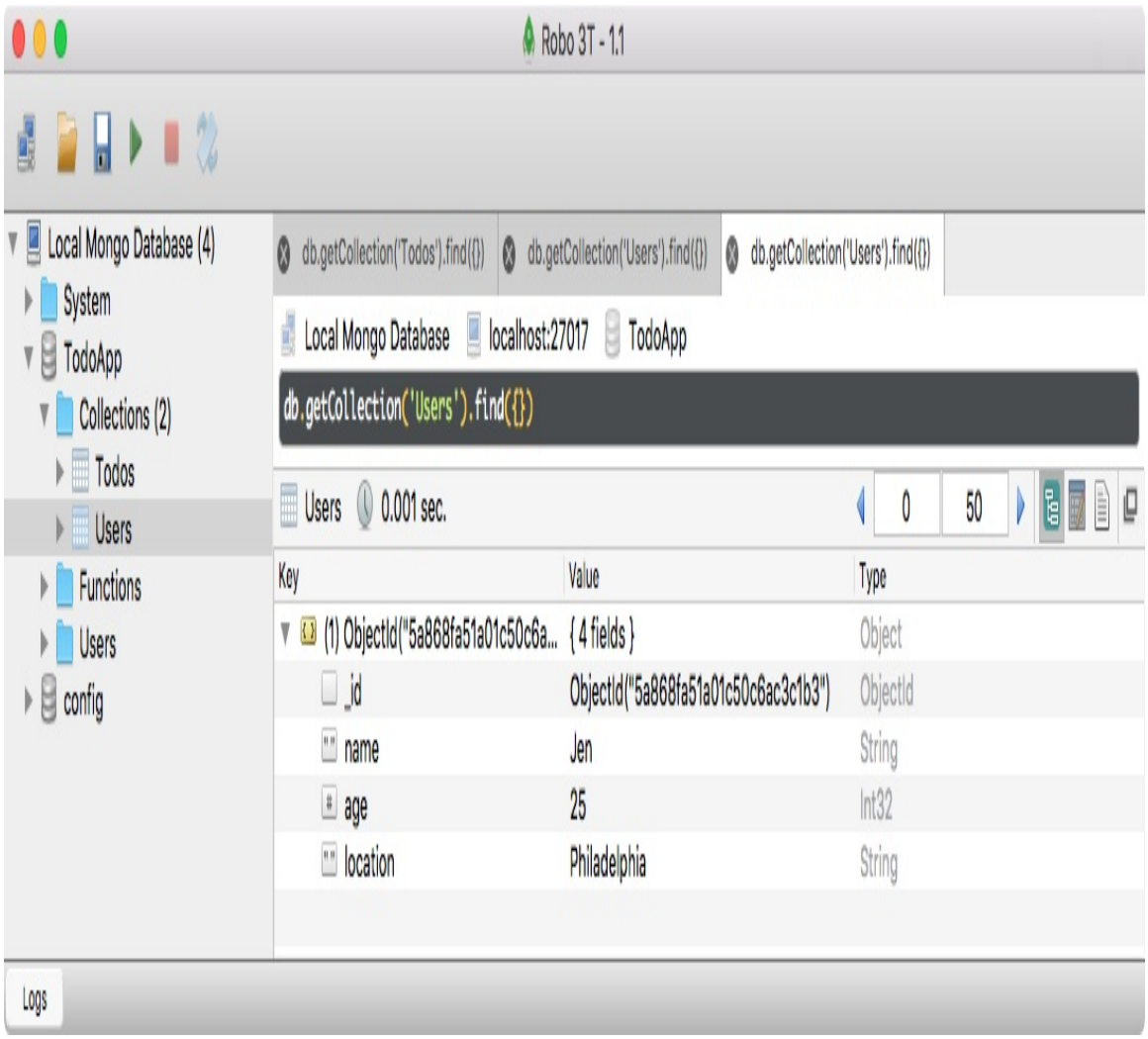
We just get the two documents back. Anything where the name was `Andrew` is now removed, which means our statement worked as expected, and this is fantastic.

Next up, we can run our `findOneAndDelete` statement. In this case, we're expecting that that one document, the one where the `name` equals `Mike`, gets removed. I'm going to go ahead and make sure I save the file. Once I do, I can move into the Terminal and rerun the script. This

time around, we get the document back where the `name` is `Mike`. We did target the correct one, and it does appear that one item was deleted:

```
Gary:node-todo-api Gary$ node playground/mongodb-delete.js
Connected to MongoDB server
{
  "lastErrorObject": {
    "n": 1
  },
  "value": {
    "_id": "5a86978929ed740ca87e5c31",
    "name": "Mike",
    "age": 25,
    "location": "Philadelphia"
  },
  "ok": 1
}
```

I can always go ahead and verify this by refreshing the collection inside of Robomongo:



I get my collection with just one document inside of it. We are now done. We know how to delete documents from our MongoDB collections; we can delete multiple documents; we can target just one, or we can target one and get its value back.

Making commit for the deleting documents methods

Before we go, let's go ahead and make a commit, pushing it up to GitHub. In the Terminal, I can shut down the script and I can run `git status` to see what files we have untracked. Here, we have our `mongodb-delete` file. I can add it using `git add .` and then I can commit, using `git commit` with the `-m` flag. Here, I can go ahead and provide a commit message, which is going to be `Add delete script`:

```
git commit -m 'Add delete script'
```

I'm going to make that commit and I am going to push it up to GitHub using `git push`, which will default to the origin remote. When you only have one remote, the first one is going to be called origin. This is the default name, just like master is the default branch. With this in place, we are now done. Our code is up on GitHub. The topic of the next section is updating, which is where you're going to learn how to update documents inside of a collection.

Updating data

You know how to insert, delete, and fetch documents out of MongoDB. In this section, you're going to learn how to update documents in your MongoDB collections. To kick things off, as usual, we're going to duplicate the last script we wrote, and we'll update it for this section.

I'm going to duplicate the `mongodb-delete` file, renaming it to `mongodb-update.js`, and this is where we'll write our update statements. I'm also going to delete all of the statements we wrote, which is the deleted data. Now that we have this in place, we can explore the one method we'll be looking at in this section. This one is called `findOneAndUpdate`. It's kind of similar to `findOneAndDelete`. It lets us update an item and get the new document back. So if I update a Todo, set it as `completed` equal to `true`, I will get that document back in the response. Now in order to get started, we're going to be updating one of the items that we have inside of our Todos collection. If I view the documents, we currently have two. The goal here is going to be to update the second item, the one where `text` equals `Eat lunch`. We're going to try to set the `completed` value to `true`, which would be a pretty common action.

If I check off a Todo item, we want to toggle that completed Boolean value. Back inside of Atom, we're going to kick things off by accessing the appropriate collection. That'll be `db.collection`. The collection name is `Todos`, and the method we'll be using is `findOneAndUpdate`. Now, `findOneAndUpdate` is going to take the most arguments we've used so far, so let's go ahead and look up the documentation for it for future reference.

Over inside of Chrome, we currently have the Cursor tab open. This is where we have the `count` method defined. If we scroll next the

Cursor tab, we have our other tabs. The one we're looking for is `Collection`. Now, inside of the Collection section, we have our typedefs and our methods. We're looking at methods here, so if I scroll down, we should be able to find `findOneAndUpdate` and click it. Now, `findOneAndUpdate` takes quite a few arguments. The first one is the `filter`. The `update` argument lets us target the document we want to update. Maybe we have the text, or most likely we have the ID of the document. Next up is the actual updates we want to make. We don't want to update the ID, we just want to filter by ID. In this case, the updates are going to be updating the completed Boolean. Then we have some options, which we are going to define. We'll use just one of them. We also have our `callback`. We're going to leave off the callback as we've been doing so so far, in favor of promises. As you can see on the documentation page, it returns a promise if no callback is passed in, and that's exactly what we expect. Let's go ahead and start filling out the appropriate arguments for `findOneAndUpdate`, kicking things off with the `filter`. What I'm going to do is filter by ID. In Robomongo, I can grab the ID of this document. I'm going to edit it and copy the ID to the clipboard. Now, in Atom, we can start querying the first object, `filter`. We're only looking for documents where the `_id` equals `new ObjectID` with the value that we copied to the clipboard. This is all we need for the `filter` argument. Next up is going to be the actual updates we want to apply, and this is not exactly straightforward. What we have to do here is learn about the MongoDB update operators.

We can view a complete list of these operators and exactly what they are by googling `mongodb update operators`. When I do this, we're looking for the `mongodb.com` documentation:

Update Operators

On this page

The following modifiers are available for use in update operations; e.g. in `db.collection.update()` and `db.collection.findAndModify()`.

Specify the operator expression in a document of the form:

```
{
  <operator1>: { <field1>: <value1>, ... },
  <operator2>: { <field2>: <value2>, ... },
  ...
}
```

Now this documentation is specific to MongoDB, which means it's going to work with all of the drivers. In this case, it is going to work with our Node.js driver. If we scroll down further, we can look at all of the update operators we have access to. The most important, and the one we're going to get started with, is the `$set` operator. This lets us set a field's value inside of our update, which is exactly what we want to do. There's other operators, like increment. This one, `$inc`, lets you increment a field's value, like the age field in our `Users` collection. Although these are super useful, we're going to get started with `$set`. In order to use one of these operators, what we need to do is type it out, `$set`, and then set it equal to an object. In this object, these are the things that we're actually going to be setting. For example, we want to set `completed` equal to `true`. If we tried to put `completed` equal to `true` at the root of the object like this, it would not work as expected. We have to use these update operators, which means we need this. Now that we have our updates in place using the set update operator, we can go ahead and provide our third and final argument. If you head over to the documentation for `findOneAndUpdate`, we can take a look at the `options` real quick. The one we care about is `returnOriginal`.

The `returnOriginal` method is defaulted to `true`, which means that it returns the original document, not the updated one, and we don't want that. When we update a document, we want to get back that updated document. What we're going to do is set `returnOriginal` to `false`, and that's going to happen in our third and final argument. This one is also going to be an object, `returnOriginal`, which is going to be setting equal to `false`.

With this in place, we are done. We can tack on a `then` call to do something with the results. I'll get my result back and I can simply print it to the screen, and we can take a look at exactly what comes back:

```
db.collection('Todos').findOneAndUpdate({
  _id: new ObjectID('5a86c378baa6685dd161da6e')
```

```
}, {
  $set: {
    completed:true
  }
}, {
  returnOriginal: false
}).then((result) => {
  console.log(result);
});
```

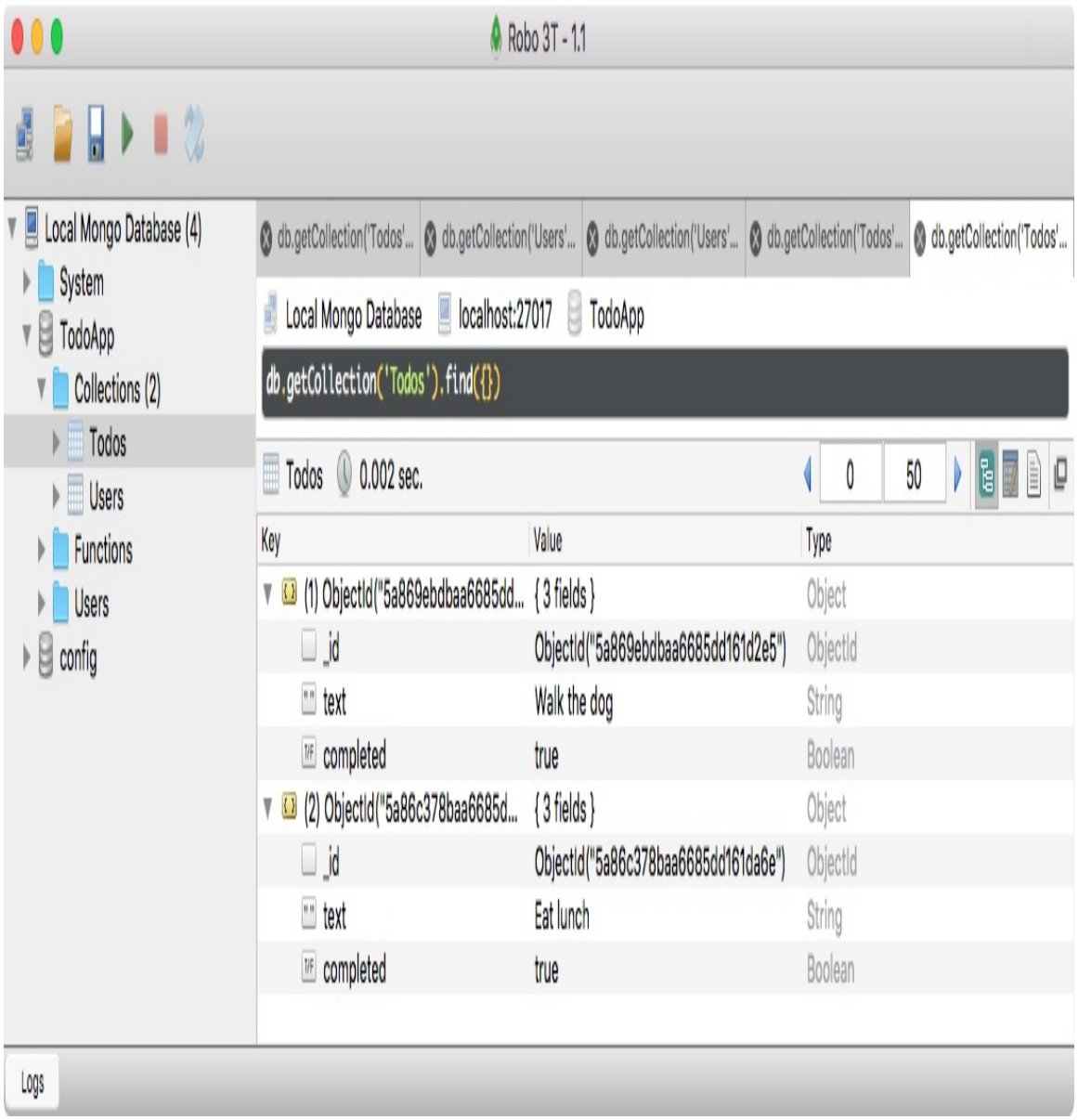
Now, let's go ahead and run this from the Terminal. I'm going to save my file inside the Terminal. We're going to be running `node`. The file is in the `playground` folder, and we will call it `mongodb-update.js`. I'm going to run the following script:

```
node playground/mongodb-update.js
```

We get back the value prop, just like we did when we used `findOneAndDelete`, and this has our document with the completed value set to true, which is the brand-new value we just set, which is fantastic:

```
Gary:node-todo-api Gary$ node playground/mongodb-update.js
Connected to MongoDB server
{ lastErrorObject: { n: 1, updatedExisting: true },
  value:
   { _id: 5a86c378baa6685dd161da6e,
     text: 'Eat lunch',
     completed: true },
  ok: 1 }
```

If we head over to Robomongo, we can confirm that the value was indeed updated. We can see this in the old document, where the value is false. I'm going to open up a new view for Todos:



We have Eat lunch, with a completed value of true. Now that we have this in place, we know how to insert, delete, update, and read documents from our MongoDB collections. To wrap this section up, I want to give you a quick challenge. Over inside of the `Users` collection, you should have a document. It should have a name. It's probably not `Jen`; it's probably something that you set. What I want you to do is update this name to your name. Now if it's already your name, that's fine; you can change it to something else. I also want you to use `$inc`, the increment operator that we talked about, to increment this by 1. Now I'm not going to tell you exactly how

increment works. What I want you to do is head over to the docs, click on the `operator`, and then scroll down to see the examples. There's examples for each operator. It's going to become really useful for you to learn how to read documentation. Now, documentation for libraries is not always going to be the same; everyone does it a little differently; but once you learn how to read the docs for one library, it gets a lot easier to read the docs for others, and I can only teach so much in this course. The real goal of this course is to get you writing your own code, doing your own research, and looking up your own documentation, so your goal once again is to update this document, setting the name to something other than what it's currently set to, and incrementing the age by 1.

To kick things off, I'm going to grab the ID of the document in Robomongo, since this is the document I want to update. I'll copy the ID to the clipboard, and now we can focus on writing that statement in Atom. First up, we'll update the name, since we already know how to do that. In Atom, I'm going to go ahead and duplicate the statement:

```
db.collection('Todos').findOneAndUpdate({
  _id: new ObjectID('57bc4b15b3b6a3801d8c47a2')
}, {
  $set: {
    completed:true
  }
}, {
  returnOriginal: false
}).then((result) => {
  console.log(result);
});
```

I'll copy it and paste it. Back inside of Atom, we can start swapping things out. First up, we're going to swap out the old ID for the new one, and we're going to change what we passed to set. Instead of updating `completed`, we want to update `name`. I'm going to set the `name` equal to something other than `Jen`. I'm going to go ahead and use my

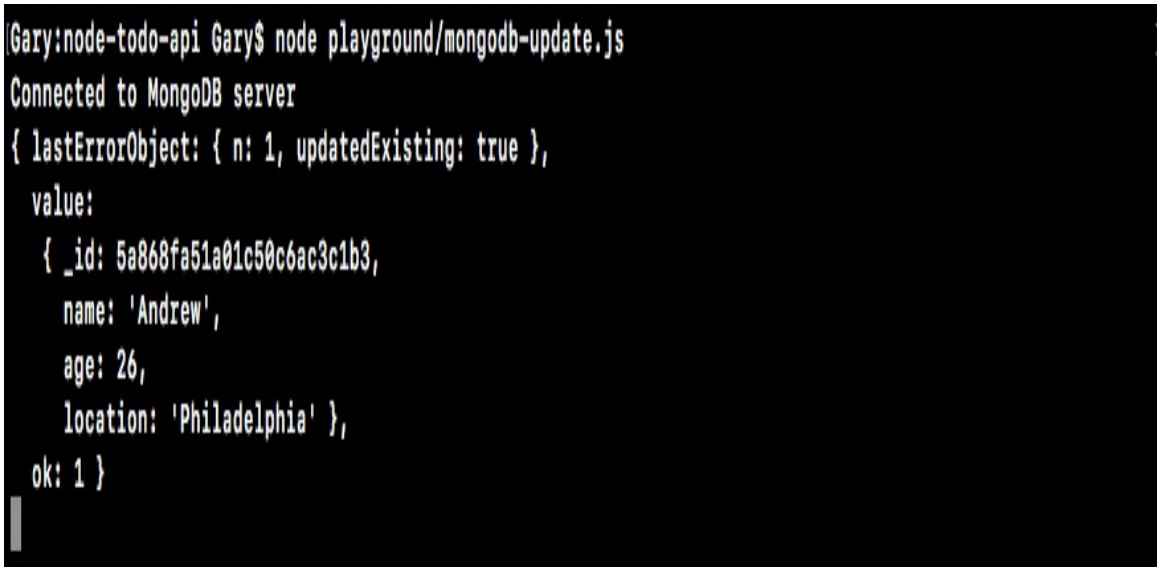
name, `Andrew`. Now, we are going to keep `returnOriginal` set to `false`. We want to get the new document back, not the original. Now, the other thing that we need to do is increment the age. This is going to be done via the increment operator, which you should have explored using the documentation over inside of Chrome. If you click on `$inc`, it's going to bring you to the `$inc` part of the documentation, and if you scroll down, you should be able to see an example. Right here, we have an example of what it looks like to increment:

We set `$inc` just like we set `set`. Then, inside of the object, we specify the things we want to increment, and the degree to which we want to increment them. It could be `-2`, or in our case, it would be positive, `1`. In Atom, we can implement this, as shown in the following code:

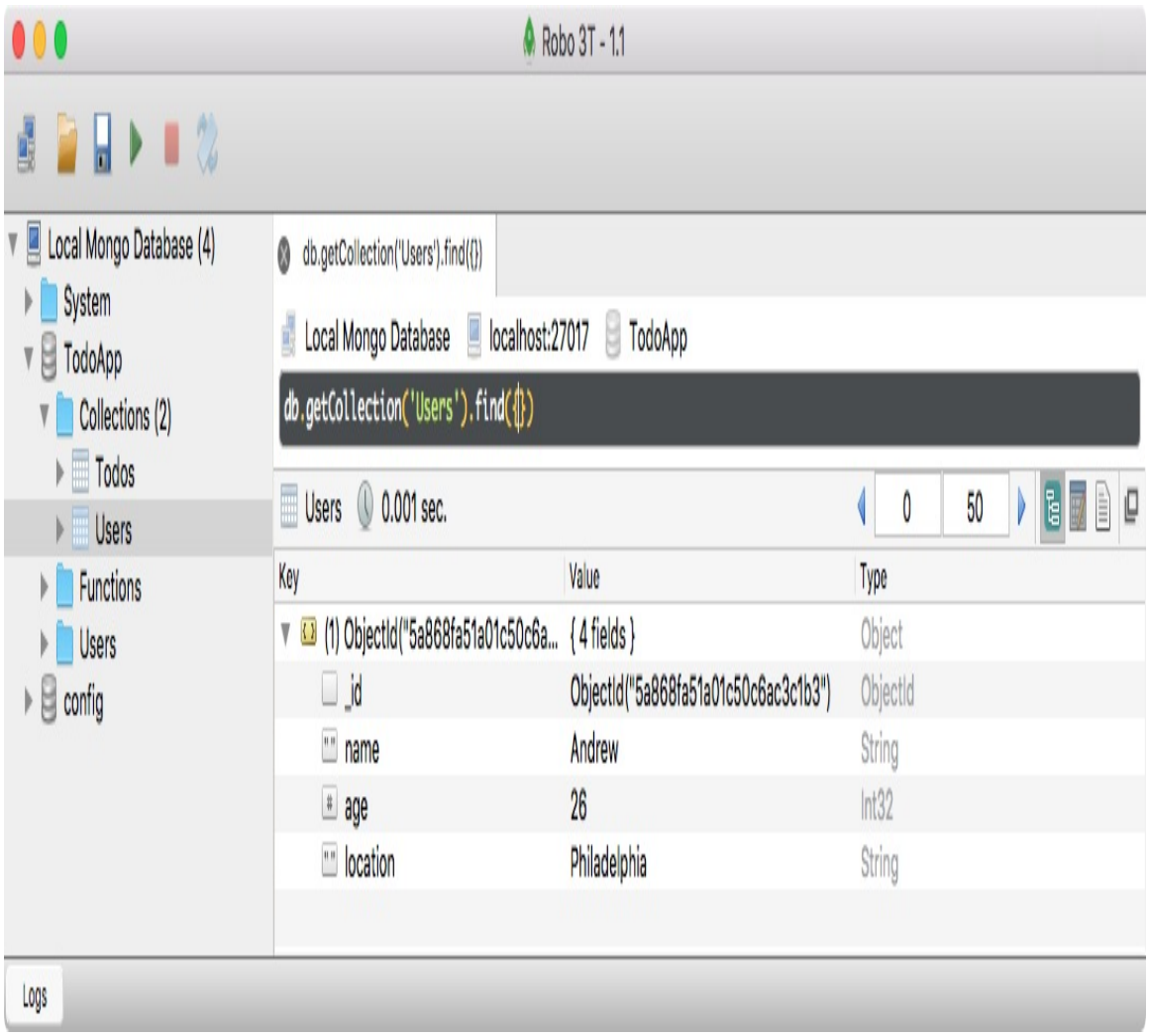
```
db.collection('Users').findOneAndUpdate({
  _id: new ObjectID('57abbcf4fd13a094e481cf2c')
}, {
  $set: {
    name: 'Andrew'
  },
  $inc: {
    age: 1
  }
}, {
  returnOriginal: false
}).then((result) => {
```

```
  console.log(result);
});
```

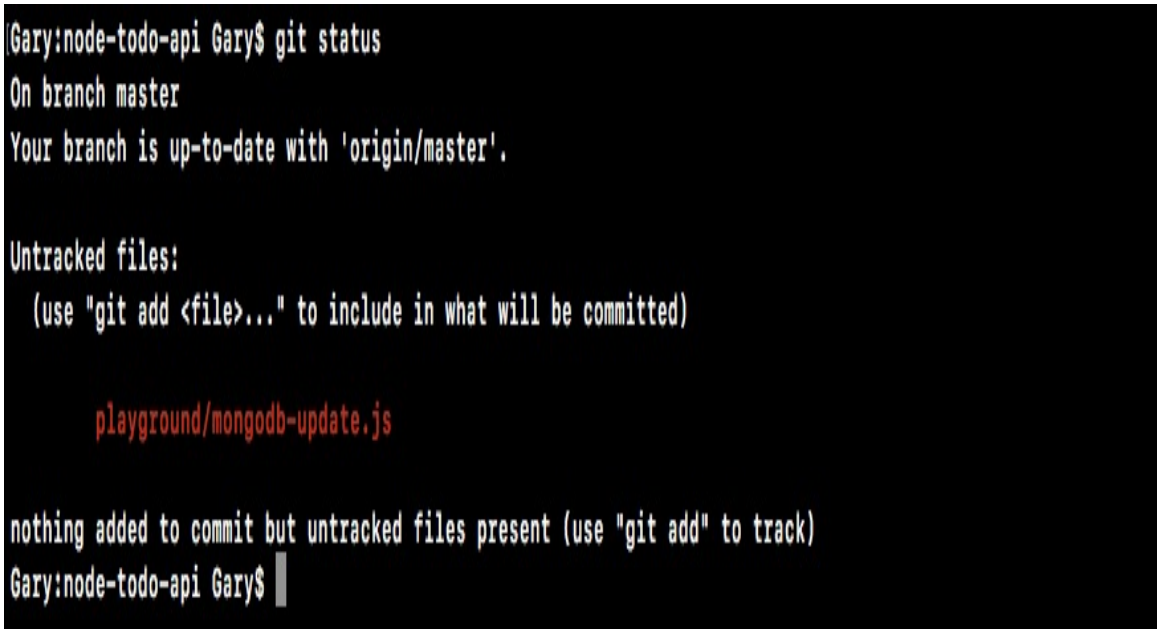
I'll set `$inc` equal to an object, and in there, we'll increment the `age` by `1`. With this in place, we are now done. Before I run this file, I am going to comment out to the other call to `findOneAndUpdate`, just leaving the new one. I also need to swap out the collection. We're no longer updating the Todos collection; we're updating the `Users` collection. Now, we are good to go. We're setting the `name` equal to `Andrew` and we're incrementing the `age` by `1`, which means that we would expect the age in Robomongo to be 26 instead of 25. Let's go ahead and run this by restarting the script over in the Terminal:



We can see our new document, where the name is indeed `Andrew` and the age is indeed `26`, and this is fantastic. Now that you know how to use the increment operator, you can also go off and learn all of the other operators you have available to you inside of your update calls. I can double-check that everything worked as expected in Robomongo. I'm going to go ahead and refresh the `Users` collection:



We have our updated document right here. Well, let's wrap this section up by committing our changes. In the Terminal, I'm going to run `git status` so we can view all of the changes to the repository:



Here, we just have one untracked file, our `mongodb-update` script. I'm going to use `git add .` to add that to the next commit, and then I'll use `git commit` to actually make the commit. I am going to provide the `-m` argument for `message` so we can specify a message, which is going to be `Add update script`:

```
git add .
git commit -m 'Add update script'
```

And now we can run the commit command and push it up to GitHub, so our code is backed up on our GitHub repository:

```
git push
```

With updating in place, we now have all of the basic CRUD (Creating, Reading, Updating, and Deleting) operations down. Up next, we're going to talk about something called Mongoose, which we'll be using for the Todo API.

Summary

In this chapter, we started with connecting to MongoDB and writing data. We then went ahead to understand the `id` property in the context of MongoDB. After learning more about fetching data, we explored different methods to delete data in the documents.

In the next chapter, we will continue to play more with Mongoose, MongoDB, and REST APIs.

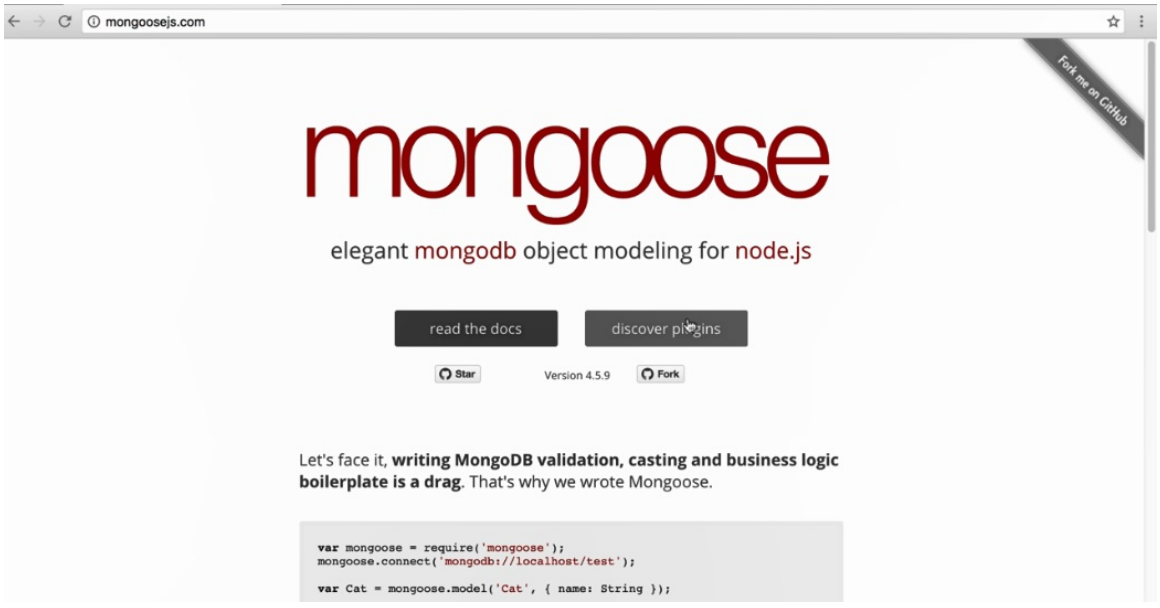
MongoDB, Mongoose, and REST APIs – Part 2

In this chapter, you're finally going to move out of the `playground` folder, and we're going to start playing with Mongoose. We'll be connecting to our MongoDB database, creating a model, talking about what exactly a model is, and finally, we'll be saving some data to the database using Mongoose.

Setting up Mongoose

We're not going to need any of the files we currently have open in the `playground` directory, so we can go ahead and close them. We're also going to wipe the `TodoApp` database using Robomongo. The data inside of Robomongo is going to be a little different than the data we'll be using going forward, and it's best to start with a clean slate. There is no need to create the database after you drop it because if you remember, MongoDB is going to automatically create the database once you start writing data to it. With this in place, we can now explore Mongoose, and the first thing I always like to do is check out the website.

You can check the website out by going to `mongoosejs.com`:



Here, you can find examples, guides, a full list of plugins, and a ton of great resources. The read the docs resource is the one I use the most. It includes tutorial-like guides that have examples, as well as documentation covering every single feature of the library. It really

is a fantastic resource.

> *If you ever want to learn about something or want to use a feature we don't cover in the book, I highly recommend coming to this page, taking the examples, copying and pasting some code, playing around with it, and figuring out how it works. We're going to be covering most of the essential Mongoose features right now.*

Setting up root of the project

The first thing we need to do before we can actually use Mongoose in our project is install it. Over in the Terminal, I'm going to install it using `npm i`, which is short for `npm install`. The module name itself is called `mongoose`, and we'll be installing the most recent version, which is going to be version `5.0.6`. We're going to tack on the `--save` flag since we will need Mongoose for both production and testing purposes:

```
npm i mongoose@5.0.6 --save
```

Once we run this command, it's going to go off and do its thing. We can move into Atom and start creating the files we're going to need to run our application.

First up, let's make a folder in the root of the project. This folder is going to be called `server`, and everything related to our server is going to get stored in the `server` folder. The first file we're going to create is going to be called `server.js`. This is going to be the root of our application. When you want to start up your Node app, you're going to run this file. This file will get everything ready to go.

The first thing we need to do inside of `server.js` is load in Mongoose. We're going to make a variable called `mongoose`, and we're going to acquire it from the `mongoose` library.

```
var mongoose = require('mongoose');
```

Now that we have the mongoose variable in place, we need to go ahead and connect to the database because we can't start writing

data to the database until Mongoose knows how to connect.

Connecting mongoose to database

The process of connecting is going to be pretty similar to what we did inside of our MongoDB scripts; for example, the `mongodb-connect` script. Here, we called `MongoClient.connect`, passing in a URL. What we're going to do for Mongoose is call `mongoose.connect`, passing in the exact same URL; `mongodb` is the protocol, call in `//`. We're going to be connecting to our `localhost` database on port `27017`. Next up is going to be our `/`, followed by the database name, and we'll continue to use the `TodoApp` database, which we used over in the `mongodb-connect` script.

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/TodoApp');
```

This is where the two functions differ. The `MongoClient.connect` method takes a callback, and that is when we have access to the database. Mongoose is a lot more complex. This is good, because it means our code can be a lot simpler. Mongoose is maintaining the connection over time. Imagine I try to save something, `save new something`. Now obviously, by the time this save statement runs, `mongoose.connect` is not going to have had time to make a database request to connect. That's going to take a few milliseconds at least. This statement is going to run almost right away.

Behind the scenes, Mongoose is going to be waiting for the connection before it ever actually tries to make the query, and this is one of the great advantages of Mongoose. We don't have to micromanage the order in which things happen; Mongoose takes care of that for us.

There is one more thing I want to configure just above `mongoose.connect`. We've been using promises in this course, and we're going to continue using them. Mongoose supports callbacks by default, but callbacks really aren't how I like to program. I prefer promises as they're a lot simpler to chain, manage, and scale. Right above the `mongoose.connect` statement, we're going to tell Mongoose which promise library we want to use. If you're not familiar with the history of promises, it didn't have to always be something built into JavaScript. Promises originally came from libraries like Bluebird. It was an idea a developer had, and they created a library. People started using it, so much so that they added it to the language.

In our case, we need to tell Mongoose that we want to use the built-in promise library as opposed to some third-party one. We're going to set `mongoose.Promise` equal to `global.Promise`, and this is something we're only going to have to do once:

```
var mongoose = require('mongoose');

mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/TodoApp');
```

We're just going to put these two lines in `server.js`; we don't have to add them anywhere else. With this in place, Mongoose is now configured. We've connected to our database and we've set it up to use promises, which is exactly what we want. The next thing we're going to do is create a model.

Creating the todo model

Now, as we have already talked about, inside of MongoDB, your collections can store anything. I could have a collection with a document that has an age property, and that's it. I could have a different document in the same collection with a property name; that's it. These two documents are different, but they're both in the same collection. Mongoose likes to keep things a little more organized than that. What we're going to do is create a model for everything we want to store. In this example, we'll be creating a Todo model.

Now, a Todo is going to have certain attributes. It's going to have a `text` attribute, which we know is a string; it's going to have a `completed` attribute, which we know is a Boolean. These are things we can define. What we're going to do is create a Mongoose model so Mongoose knows how to store our data.

Right below the `mongoose.connect` statement, let's make a variable called `Todo`, and we're going to set that equal to `mongoose.model`. The `model` is the method we're going to use to create a new model. It takes two arguments. The first one is the string name. I'm going to match the variable name on the left, `Todo`, and the second argument is going to be an object.

```
mongoose.connect('mongodb://localhost:27017/TodoApp');
var Todo = mongoose.model('Todo', {

});
```

This object is going to define the various properties for a model. For example, the Todo model is going to have a `text` property, so we can set that up. Then, we can set text equal to an object, and we can

configure exactly what text is. We can do the same thing for `completed`. We're going to have a completed property, and we're going to want to specify certain things. Maybe it's required; maybe we have custom validators; maybe we want to set the type. We're also going to add one final one, `completedApp`, and this is going to let us know when a Todo was completed:

```
var Todo = mongoose.model('Todo', {
  text: {

  },
  completed: {

  },
  completedAt: {

  }
});
```

A `createdApp` property might sound useful, but if you remember the MongoDB `ObjectId`, that already has the `createdAt` timestamp built in, so there's no reason to add a `createdApp` property here. `completedAt`, on the other hand, is going to add value. It lets you know exactly when you have completed a Todo.

From here, we can start specifying the details about each attribute, and there's a ton of different options available inside of the Mongoose documentation. For now though, we're going to keep things really simple by specifying the type for each, for example, `text`. We can set `type` equal to `String`. It's always going to be a string; it wouldn't make sense if it was a Boolean or a number.

```
var Todo = mongoose.model('Todo', {
  text: {
    type: String
  },
```

Next, we can set a type for `completed`. It needs to be a Boolean; there's

no way around that. We're going to set `type` equal to `Boolean`.

```
  completed: {
    type: Boolean
  },
```

The last one we have is `completedAt`. This is going to be a regular old Unix timestamp, which means it's just a number, so we can set the `type` for `completedAt` equal to `Number`:

```
  completedAt: {
    type: Number
  }
});
```

With this in place, we now have a working Mongoose model. It's a model of a Todo that has a few properties: `text`, `completed`, and `completedAt`.

Now in order to illustrate exactly how we create instances of this, we're going to go ahead and just add one Todo. We're not going to worry about fetching data, updating data, or deleting data, although that is stuff that Mongoose supports. We'll be worrying about that in the following sections, as we start building out the individual routes for our API. For now, we're going to go over just a very quick example of creating a brand-new Todo.

Creating a brand-new Todo

I'm going to make a variable called `newTodo`, although you could call it anything you like; the name here is not important. What is important though is that you run the Todo function. This is what comes back from `mongoose.model` as a constructor function. We want to add the `new` keyword in front of it because we're creating a new instance of `Todo`.

Now, the `Todo` constructor function does take an argument. It's going to be an object where we can specify some of these properties. Maybe we know that we want `text` to equal something like `Cook dinner`. Right in the function, we can specify that. `text` equals a string, `Cook dinner`:

```
var newTodo = new Todo({
  text: 'Cook dinner'
});
```

We haven't required any of our attributes, so we could just stop here. We have a `text` property; that's good enough. Let's go ahead and explore how to save this to the database.

Saving the instance to the database

Creating a new instance alone does not actually update the MongoDB database. What we need to do is call a method on `newTodo`. This is going to be `newTodo.save`. The `newTodo.save` method is going to be responsible for actually saving `text` to the MongoDB database. Now, `save` returning a promise, which means we can tack on a `then` call and add a few callbacks.

```
newTodo.save().then((doc) => {

}, (e) => {

});
```

We'll add the callbacks for when the data either gets saved or when an error occurs because it can't save for some reason. Maybe the connection failed, or maybe the model is not valid. Either way, for now we'll just print a little string, `console.log(Unable to save todo)`. Up above, in the success callback, we're actually going to get that Todo. I can call the argument `doc`, and I can print it to the screen, `console.log`. I'll print a little message first: `Saved todo`, and the second argument will be the actual document:

```
newTodo.save().then((doc) => {
  console.log('Saved todo', doc);
}, (e) => {
  console.log('Unable to save todo');
});
```

We've configured Mongoose, connecting to the MongoDB database;

we've created a model, specifying the attributes we want Todos to have; we created a new Todo; and finally, we saved it to the database.

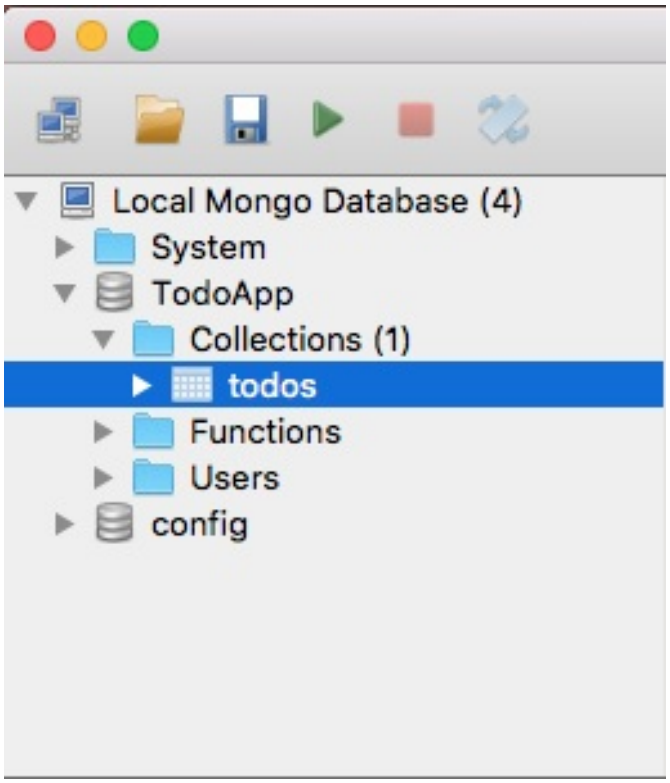
Running the Todos script

We're going to run the script from the Terminal. I'm going to kick things off by running `node`. The file we're running is in the `server` directory, and it's called `server.js`:

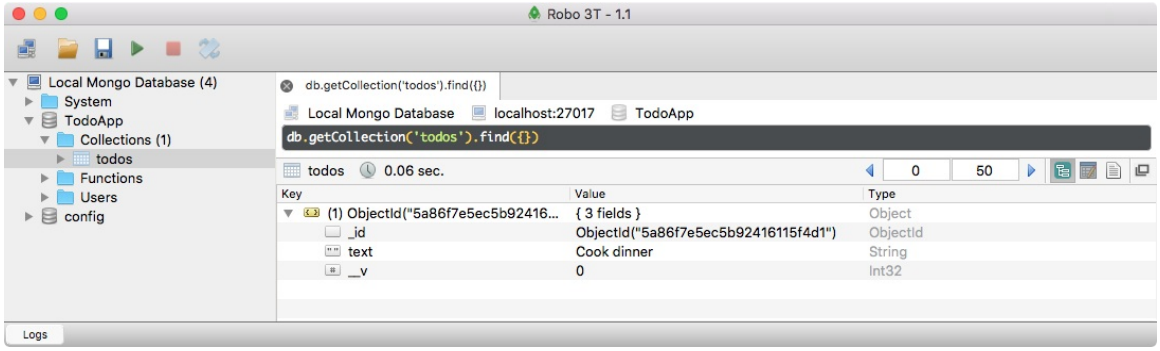
```
node server/server.js
```

When we run the file, we get `Saved todo`, meaning that things went well. We have an object right here with an `_id` property as expected; the `text` property, which we specified; and the `__v` property. The `__v` property means version, and it comes from Mongoose. We'll talk about it later, but essentially it keeps track of the various model changes over time.

If we open up Robomongo, we're going to see the exact same data. I'm going to right-click the connection and refresh it. Here, we have our `TodoApp`. Inside of the `TodoApp` database, we have our `todos` collection:



Notice that Mongoose automatically lowercased and pluralized Todo. I'm going to view the documents:



We have our one document with the text equal to Cook dinner, exactly what we created over inside of Atom.

Creating a second Todo model

We have one Todo created using our Mongoose model. What I want you to do is make a second one, filling out all three values. This means you're going to make a new Todo with a `text` value, a `completed` Boolean; go ahead and set that to `true`; and a `completedAt` timestamp, which you can set to any number you like. Then, I want you to go ahead and save it; print it to the screen if it saves successfully; print an error if it saves poorly. Then, finally, run it.

The first thing I would have done is made a new variable down below. I'm going to make a variable called `otherTodo`, setting it equal to a `new` instance of the `Todo` model.

```
var otherTodo = new Todo ({

});
```

From here, we can pass in our one argument, which is going to be the object, and we can specify all of these values. I can set `text` equal to whatever I like, for example, `Feed the cat`. I can set the `completed` value equal to `true`, and I can set `completedAt` equal to any number. Anything lower than 0, like -1, is going to go backwards from 1970, which is where 0 is. Anything positive is going to be where we're at, and we'll talk about time-stamps more later. For now, I'm going to go with something like `123`, which would basically be two minutes into the year 1970.

```
var otherTodo = new Todo ({
  text: 'Feed the cat',
  completed: true,
  completedAt: 123
});
```

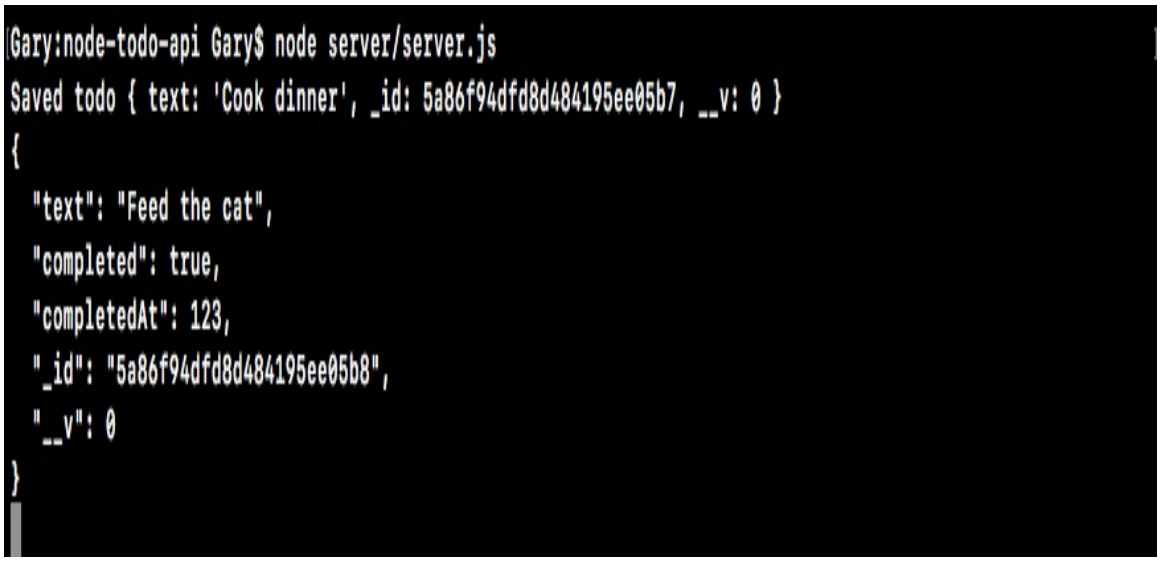
With this in place, we now just need to call `save`. I'm going to call `otherTodo.save`. This is what's actually going to write to the MongoDB database. I am going to tack on a `then` callback, because I do want to do something once the save is complete. If the `save` method worked, we're going to get our `doc`, and I'm going to print it to the screen. I'm going to use that pretty-print system we talked about earlier, `JSON.stringify`, passing in the actual object, `undefined`, and `2`.

```
var otherTodo = new Todo ({
  text: 'Feed the cat',
  completed: true,
  completedAt: 123
});

otherTodo.save().then((doc) => {
  console.log(JSON.stringify(doc, undefined, 2));
})
```

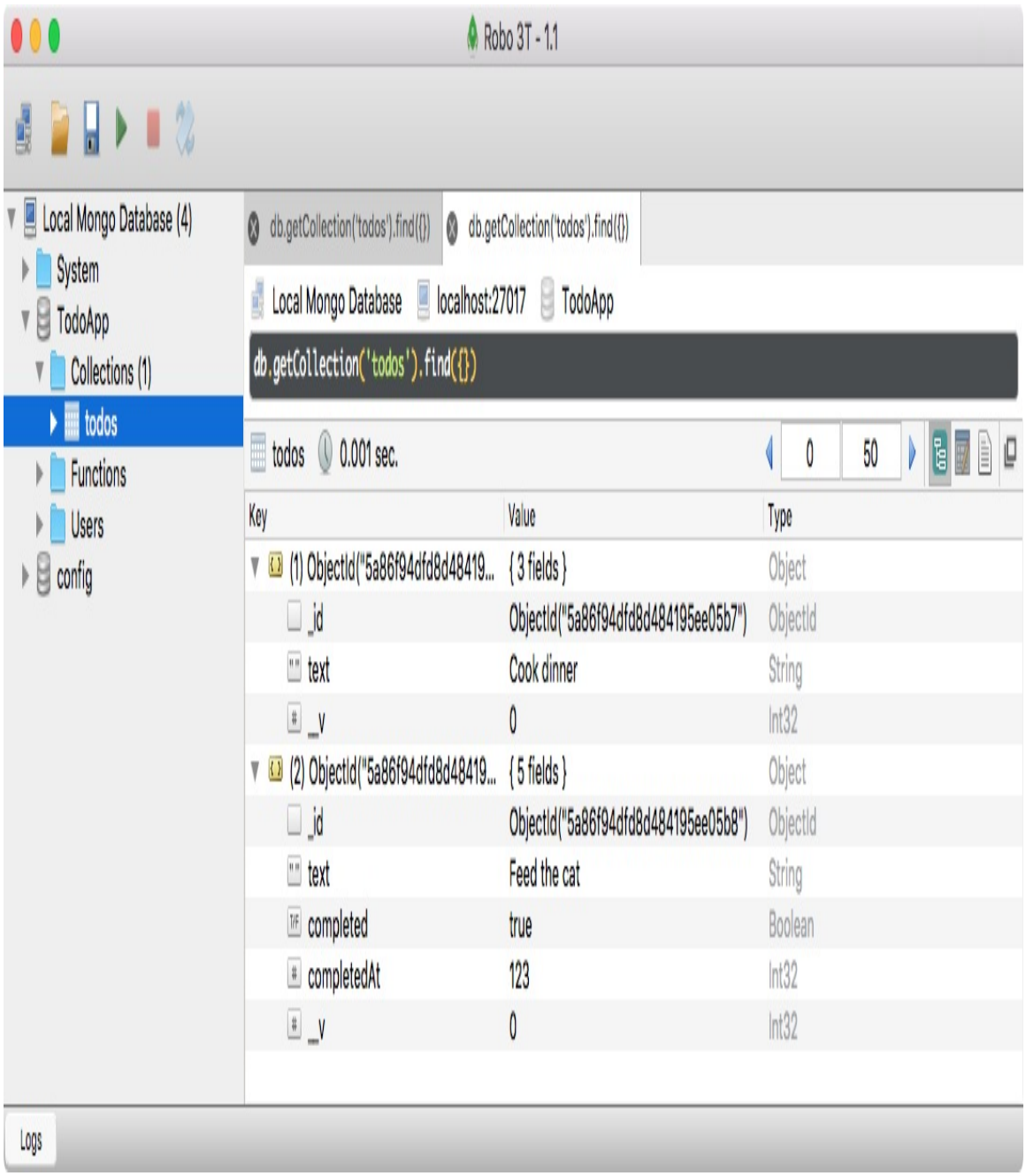
You don't need to do this; you can print it in any way you like. Next up, I'm going to print a little message if things go poorly: `console.log('Unable to save', e)`. It'll pass along that error object, so if someone's reading the logs, they can see exactly why the call failed:

```
otherTodo.save().then((doc) => {
  console.log(JSON.stringify(doc, undefined, 2));
}, (e) => {
  console.log('Unable to save', e);
});
```

With this in place, we can now comment out that first Todo. This is going to prevent another one from being created, and we can rerun the script, running our brand-new Todo creation calls. In the Terminal, I'm going to shut down the old connection and start up a new one. This is going to create a brand-new Todo, and we have it right here:



The `text` property equals `Feed the cat`. The `completed` property sets to the Boolean `true`; notice there's no quotes around it. The `completedAt` equals the number `123`; once again, no quotes. I can also go into Robomongo to confirm this. I'm going to refetch the Todos collection, and now we have two Todos:



On the right-hand side of the Values column, you'll also notice the Type column. Here, we have int32 for completedAt and the __v property. The completed property is a Boolean, text is a String, and the _id is an ObjectId type.

There's a lot of useful information hidden inside of Robomongo. If you want something, they most likely have it built in. That's it for this one. We now know how to use Mongoose to make a connection,

create a model, and finally save that model to the database.

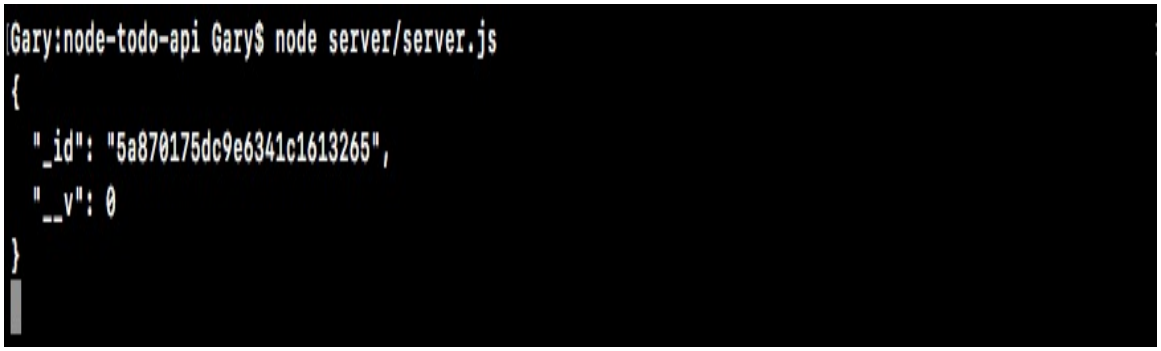
Validators, Types, and Defaults

In this section, you're going to learn how to improve your Mongoose models. This is going to let you add things like validation. You can make certain properties be a requirement, and you can set up smart defaults. So, if something like completed is not provided, you can have a default value that gets set. All of this functionality is built into Mongoose; we just have to learn how to use it.

To illustrate why we'd want to set this stuff up, let's scroll to the bottom of our `server` file and remove all of the properties on the `new Todo` we created. Then, we're going to save the file and move into the Terminal, running the script. That's going to be `node` in the `server` directory, and the file is going to be called `server.js`:

```
node server/server.js
```

When we run it, we get our new Todo, but it only has the version and ID properties:



All of the properties we specified in the model, `text`, `completed`, and `completedAt`, are nowhere to be found. That's a pretty big problem. We should not be adding Todos to the database if they don't have a `text` property, and things like `completed` should have smart defaults. No-

one's going to create a Todo item if they already completed it, so completed should default to `false`.

Mongoose validators

Now in order to get started, we're going to pull up two pages in the Mongoose documentation, just so you know where this stuff lives if you ever want to dive deeper in the future. First up, we're going to look up the validators. I'm going to google `mongoose validators`, and this is going to show us all of the default validation properties we have built in:

Validation

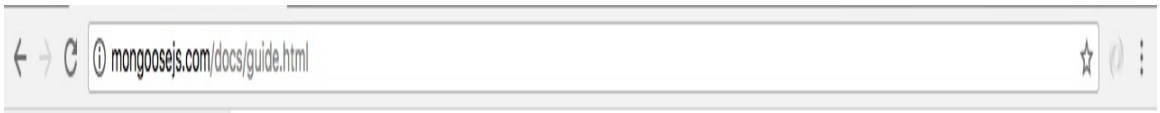
Before we get into the specifics of validation syntax, please keep the following rules in mind:

- Validation is defined in the SchemaType
- Validation is middleware. Mongoose registers validation as a `pre('save')` hook on every schema by default.
- You can manually run validation using `doc.validate(callback)` or `doc.validateSync()`
- Validators are not run on undefined values. The only exception is the `required` validator.
- Validation is asynchronously recursive; when you call Model#save, sub-document validation is executed as well. If an error occurs, your Model#save callback receives it
- Validation is customizable

```
var schema = new Schema({
  name: {
    type: String,
    required: true
  }
});
var Cat = db.model('Cat', schema);

// This cat has no name :(
var cat = new Cat();
cat.save(function(error) {
  assert.equal(error.errors['name'].message,
    'Path `name` is required.');
```

For example, we can set something as `required`, so if it's not provided it's going to throw an error when we try to save that model. We can also set up validators for things like numbers and strings, giving a `min` and `max` value or a `minlength`/`maxlength` value for a string.

The other page we're going to look at is the Schemas page. To get to this, we're going to google `mongoose schemas`. This is the first one, the `guide.html` file:



Schemas

If you haven't yet done so, please take a minute to read the quickstart to get an idea of how Mongoose works. If you are migrating from 4.x to 5.x please take a moment to read the migration guide.

Defining your schema

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

If you want to add additional keys later, use the Schema#add method.

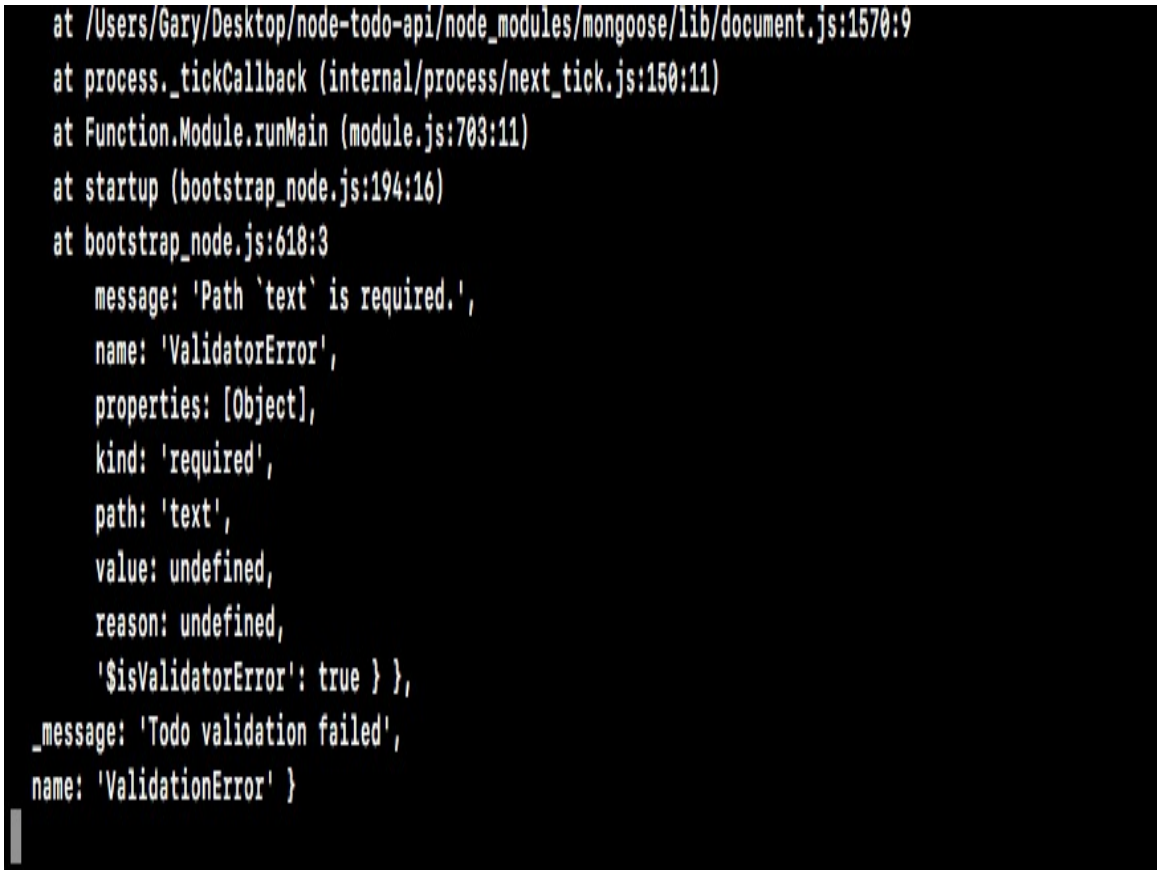
On this page, you're going to see something slightly different from what we've been doing so far. They call `new Schema`, setting up all of their properties. This is not something we've done yet, but we will in the future. For now, you can consider this object, the `Schema` object, identical to the one we have over in Atom that we pass in as the second argument to our `mongoose.model` call.

Customizing the Todo text property

To kick things off, let's customize how Mongoose treats our `text` property. Currently, we tell Mongoose that we want it to be a string, but we don't have any validators. One of the first things we can do for the `text` property is set `required` equal to `true`.

```
var Todo = mongoose.model('Todo', {
  text: {
    type: String,
    required: true
  },
```

When you set `required` equal to `true`, the value must exist, so if I were to try to save this Todo it would fail. And we can prove this. We can save the file, head over to the Terminal, shut things down, and restart it:



We get an unreadable error message. We'll dive into this in a second, but for now all you need to know is that we're getting a validation error: Todo validation failed, and that is fantastic.

Now, aside from just making sure the `text` property exists, we can also set up some custom validators. For strings, for example, we have a `minlength` validator, which is great. You shouldn't be able to create a Todo whose text is an empty string. We can set `minlength` equal to the minimum length, which we're is going to be `1` in this case:

```
var Todo = mongoose.model('Todo', {
  text: {
    type: String,
    required: true,
    minlength: 1
  },
```

Now, even if we do provide a `text` property in the `otherTodo` function, let's say we set `text` equal to an empty string:

```
var otherTodo = new Todo ({
  text: ''
});
```

It's still going to fail. It is indeed there but it does not pass the minlength validator, where the `minlength` validator must be `1`. I can save the `server` file, restart things over in the Terminal, and we still get a failure.

Now aside from `required` and `minlength`, there are a couple other utilities that are around in the docs. One good example is something called `trim`. It's fantastic for strings. Essentially, `trim` trims off any white space in the beginning or end of your value. If I set `trim` equal to `true`, like this:

```
var Todo = mongoose.model('Todo', {
  text: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
```

It's going to remove any leading or trailing white space. So if I try to create a Todo whose `text` property is just a bunch of spaces, it's still going to fail:

```
var otherTodo = new Todo ({
  text: '      '
});
```

The `trim` property is going to remove all of the leading and trailing spaces, leaving an empty string, and if I rerun things, we still get a

failure. The text field is invalid. If we do provide a valid value, things are going to work as expected. Right in the middle of all of the spaces in `otherTodo`, I'm going to provide a real Todo value, which is going to be `Edit this video`:

```
var otherTodo = new Todo ({
  text: '    Edit this video    '
});
```

When we try to save this Todo, the first thing that's going to happen is the spaces in the beginning and the end of the string are going to get trimmed. Then, it's going to validate that this string has a minimum length of 1, which it does, and finally, it will save the Todo to the database. I'm going to go ahead and save `server.js`, restart our script, and this time around we get our Todo:

```
Gary:node-todo-api Gary$ node server/server.js
{
  "text": "    Edit this video    ",
  "_id": "5a8704bfbc6d00424c6ef266",
  "__v": 0
}
```

The `Edit this video` text shows up as the `text` property. Those leading and trailing spaces have been removed, which is fantastic. Using just three properties, we were able to configure our `text` property, setting up some validation. Now, we can do similar stuff for `completed`.

Mongoose defaults

For `completed`, we're not going to `require` it because the completed value is most likely going to default to `false`. What we can do instead is set the `default` property, giving this `completed` field a default value.

```
completed: {
  type: Boolean,
  default: false
},
```

Now `completed`, as we talked about earlier in the section, should default to `false`. There's no reason to create a Todo if it's already done. We can do the same thing for `completedAt`. If a Todo starts off not completed, then `completedAt` is not going to exist. It is only going to exist when the Todo has been completed; it's going to be that timestamp. What I'm going to do is set `default` equal to `null`:

```
completed: {
  type: Boolean,
  default: false
},
completedAt: {
  type: Number,
  default: null
}
```

Awesome. Now, we have a pretty good schema for our Todo. We're going to validate that the text is set up properly by the user, and we are going to set up the `completed` and `completedAt` values by our-self since we can just use defaults. With this in place, I can now rerun our `server` file, and here we get a better default Todo:

We have the `text` property and the user provided, which has been validated and trimmed. Next, we have `completed` set to `false` and `completedAt` set to `null`; this is fantastic. We now have a foolproof schema that has good defaults and validation.

Mongoose types

If you've been playing around with the various types, you might have noticed that if you set a `type` equal to something other than the type you specified, in certain cases it does still work. For example, if I try to set `text` equal to an object, I'm going to get an error. It's going to say hey, you tried to use a string, but an object showed up instead. However, if I try to set `text` equal to something like a number, I'm going to go with `23`:

```
var otherTodo = new Todo ({
  text: 23
});
```

This is going to work. That's because Mongoose is going to cast your number into a string, essentially wrapping it in quotes. The same thing is going to be true with the Boolean. If I pass in a Boolean like this:

```
var otherTodo = new Todo ({
  text: true
});
```

The resulting string is going to be `"true"`. I'm going to go ahead and save the file after setting `text` equal to `true`, and run the script:

```
Gary:node-todo-api Gary$ node server/server.js
{
  "text": "true",
  "completed": false,
  "completedAt": null,
  "_id": "5a8706b30fe427425be3d99e",
  "__v": 0
}
```

When I do it, I get `text` equal to `true`, as shown in the preceding screenshot. Notice it is indeed wrapped in quotes. It's important to be aware that typecasting does exist inside of Mongoose. It can easily trip you up and cause some unexpected errors. For now though, I am going to set `text` equal to a proper string:

```
var otherTodo = new Todo ({
  text: 'Something to do'
});
```

Creating a Mongoose user model for authentication

Now, we're going to create a brand-new Mongoose model. First up, you're going to make a new `User` model. Eventually, we're going to use this for authentication. It's going to store stuff like an email and a password, and the Todos are going to be associated with that `User` so when I create one, only I can edit it.

We'll look into all these, but for now, we're going to keep things really simple. On the `User` model, the only property that you need to set up is the `email` property. We'll set up others like `password` later, but it's going to be done a little differently since it needs to be secure. For now, we'll just stick with `email`. I want you to `require` it. I also want you to `trim` it, so if someone adds spaces before or after, those spaces go away. Last but not least, go ahead and set the `type` equal to a `String`, set type, and set `minlength` of `1`. Now, obviously, you'll be able to pass in a string that's not an email. We'll explore custom validation a little later. This is going to let us validate that the email is an email, but for now this is going to get us on the right track.

Once you have your Mongoose model created, I want you to go ahead and try to create a new `User`. Create one without the `email` property, and then make one with the email property, making sure that when you run the script, the data shows up as expected over in Robomongo. This data should show up in the new `Users` collection.

Setting up the email property

The first thing I'm going to do is make a variable to store this new model, a variable called `User`, and I'm going to set that equal to `mongoose.model`, which is how we can make our new `User` model. The first argument, as you know, needs to be the string model name. I'm going to use the exact same name as I specified over in the variable, although it could be different. I just like to keep things using this pattern, where the variable equals the model name. Next up, as the second argument, we can specify the object where we configure all the properties a `User` should have.

```
var User = mongoose.model('User', {

});
```

Now as I mentioned previously, we'll be adding others later, but for now, adding support for an `email` property will be good enough. There's a few things I want to do on this email. First up, I want to set the `type`. An email is always going to be a string, so we can set that `type` equal to `String`.

```
var User = mongoose.model('User', {
  email: {
    type: String,

  }
});
```

Next up, we're going to `require` it. You can't make a user without an email, so I'll set `required` equal to `true`. After required, we're going to go ahead and `trim` that email. If someone adds spaces before or after

it, it's clearly a mistake, so we'll go ahead and remove those for the `User` model, making our application just a little more user-friendly. Last but not least, what we want to do is set up a `minlength` validator. We'll be setting up custom validation later, but for now `minlength` of `1` is going to get the trick done.

```
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1
  }
});
```

Now, I am going to go ahead and create a new instance of this `User` and save it. Before I run the script though, I will be commenting out our new Todo. Now, we can make a new instance of this `User` model. I'm going to make a variable called `user` and set it equal to `new User`, passing in any values we want to set on that user.

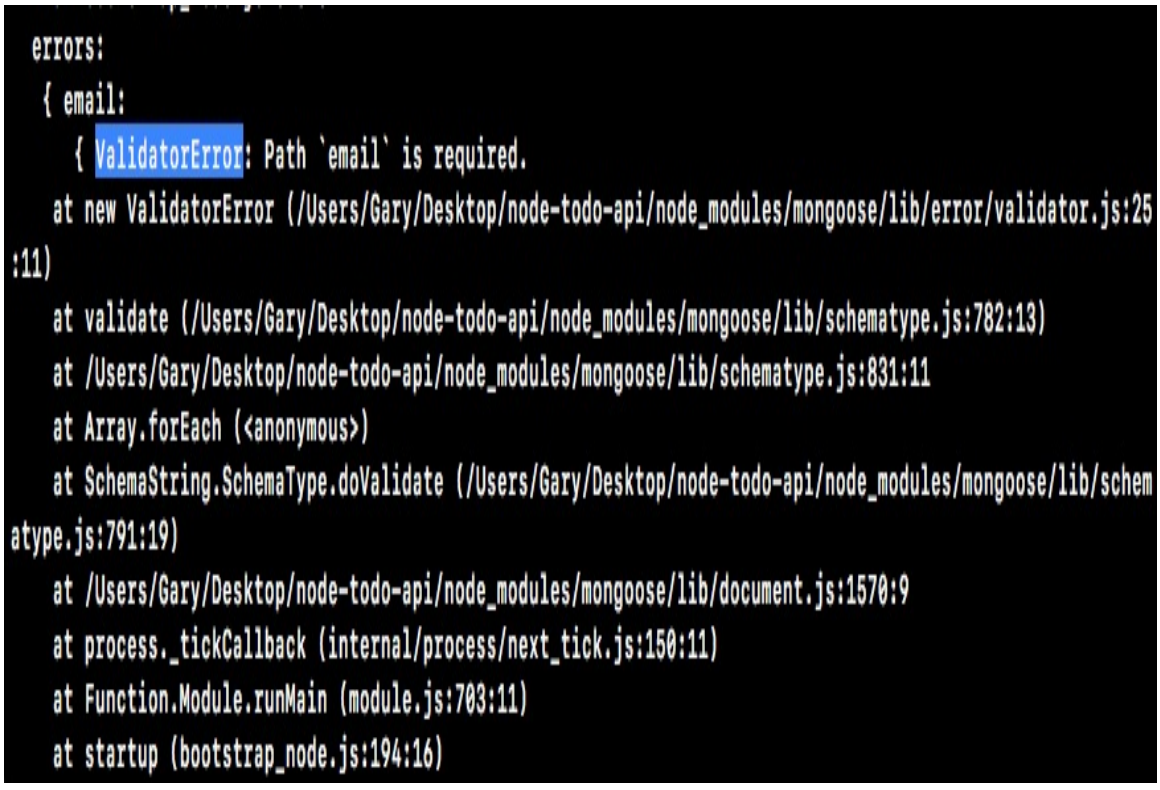
```
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1
  }
});

var user = new User({

});
```

I'm going to run it with nothing at first, just to make sure the validation is working. Next to the user variable, I can now call `user.save`. The `save` method returns a promise, so I can tack on a `then` callback. I'm going to add a success case for this one, and an error handler. The error handler will get that error argument, and the

success case will get the doc. If things go well, I'll print a message using `console.log('User saved', doc)`, followed by the `doc` argument. No need to format it for this example. I'll do the same thing for the error handler, using `console.log('Unable to save user')` followed by the error object:

```
var user = new User({

});

user.save().then((doc) => {
  console.log('User saved', doc);
}, (e) => {
  console.log('Unable to save user', e);
});
```

Since we're creating a user with no properties, we would expect the error to print. I'm going to save `server.js` and restart the file:

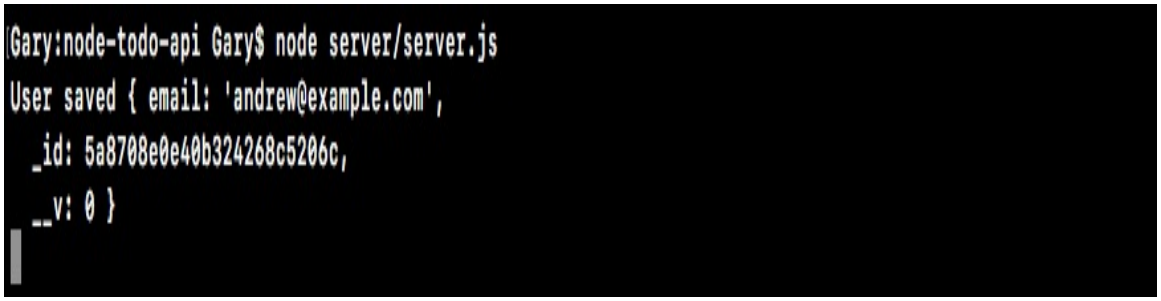


We get our error. It's a validation error called Path 'email' is

required. Mongoose is letting us know that we do indeed have an error. The email does need to exist, since we set `required` equal to `true`. I'm going to go ahead and put a value, setting `email` to my email, `andrew@example.com`, and I'll put a few spaces afterwards:

```
var user = new User({
  email: 'andrew@example.com '
});
```

This time around, things should go as expected and `trim` should be trimming the end of that email, removing all of the spaces, and that's exactly what we get:



The `User` was indeed saved, which is great, and the `email` has been properly formatted. Now obviously, I could have put a string in like `123`, and it would have worked because we don't have custom validation set up just yet, but we have a pretty good starting point. We have the `User` model, and we have our `email` property set up and ready to go.

With this in place, we are now going to start creating the API. In the next section, you're going to install a tool called **Postman**, which is going to help us test our HTTP requests, and then we're going to create our very first route for our Todo REST API.

Installing Postman

In this section, you're going to learn how to use Postman. Postman is an essential tool if you're building a REST API. I have never worked with a team or on a project where Postman was not heavily used by every developer involved. Postman lets you create HTTP requests and fire them off. This makes it really easy to test that everything you're writing is working as expected. Now obviously, we will also be writing automated tests, but using Postman lets you play around with data and see how things work as you move through your API. It really is a fantastic tool.

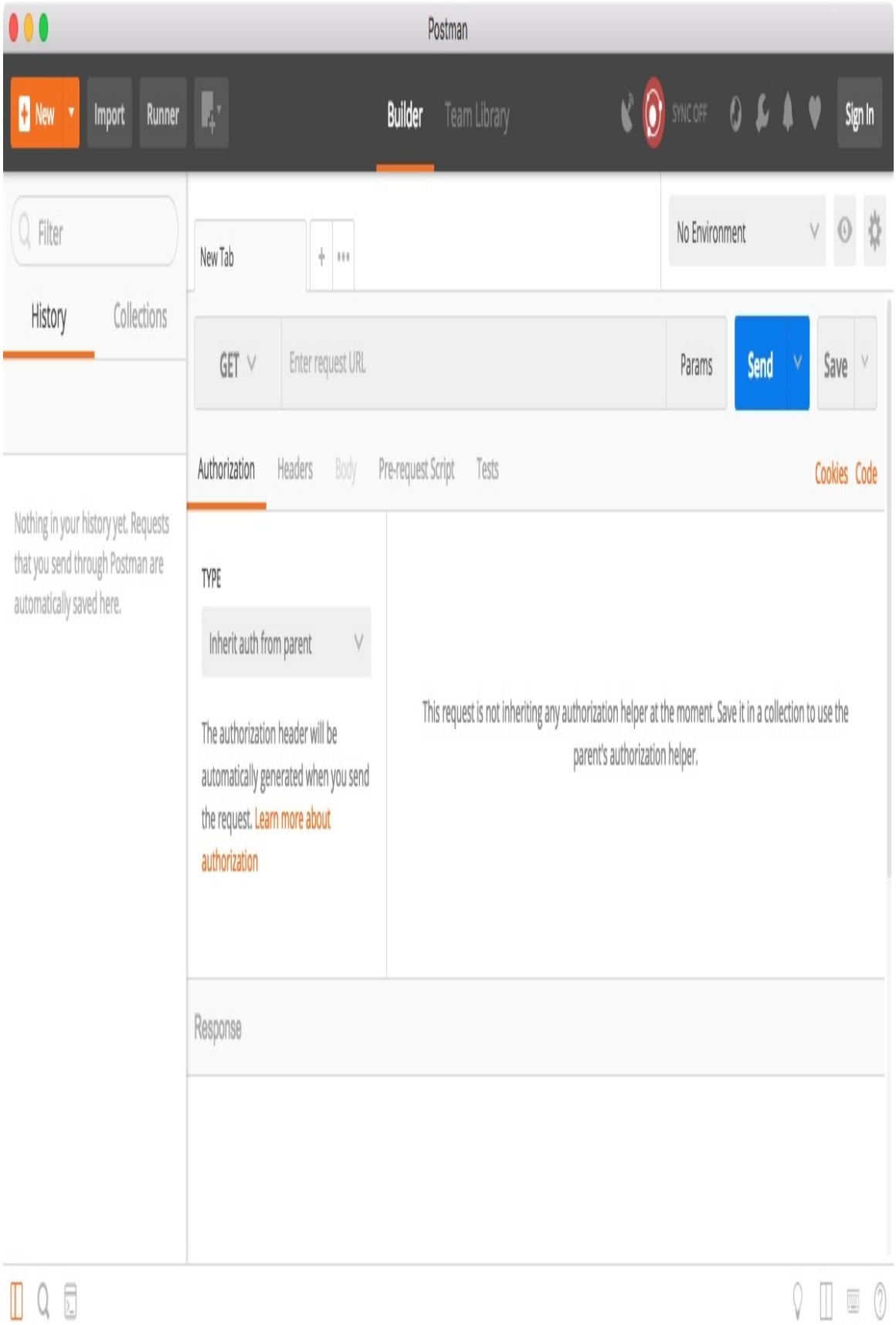
We're going to head over to the browser and go to `getpostman.com`, and here we can grab their application:


Secure | https://www.getpostman.com
Products
Solutions
Pricing
Docs
API Network
About
Sign in
POSTMAN
Developing APIs is hard
Postman makes it easy
Download the free Postman App:
macOS
Windows
Linux

Now I'm going to be using the Chrome application. To install it, all you have to do is install the Chrome app from the Chrome store, click Add to Chrome, and it should bring you over to the page where you can open up the application. Now, to open up Chrome apps, you have to go to this kind of weird URL. It's `chrome://apps`. Here, you can view all of your apps, and we can just open up Postman by clicking it.

Now as I mentioned previously, Postman lets you make HTTP requests, so we're going to go ahead and make a few to play around with the user interface. You do not need to make an account, and you do not need to sign up for a paid plan. The paid plans are targeted towards teams of developers who need advanced features. We are just making basic requests on our machine; we don't need cloud storage or anything like that. I'm going to skip account creation, and we can go right to the application.

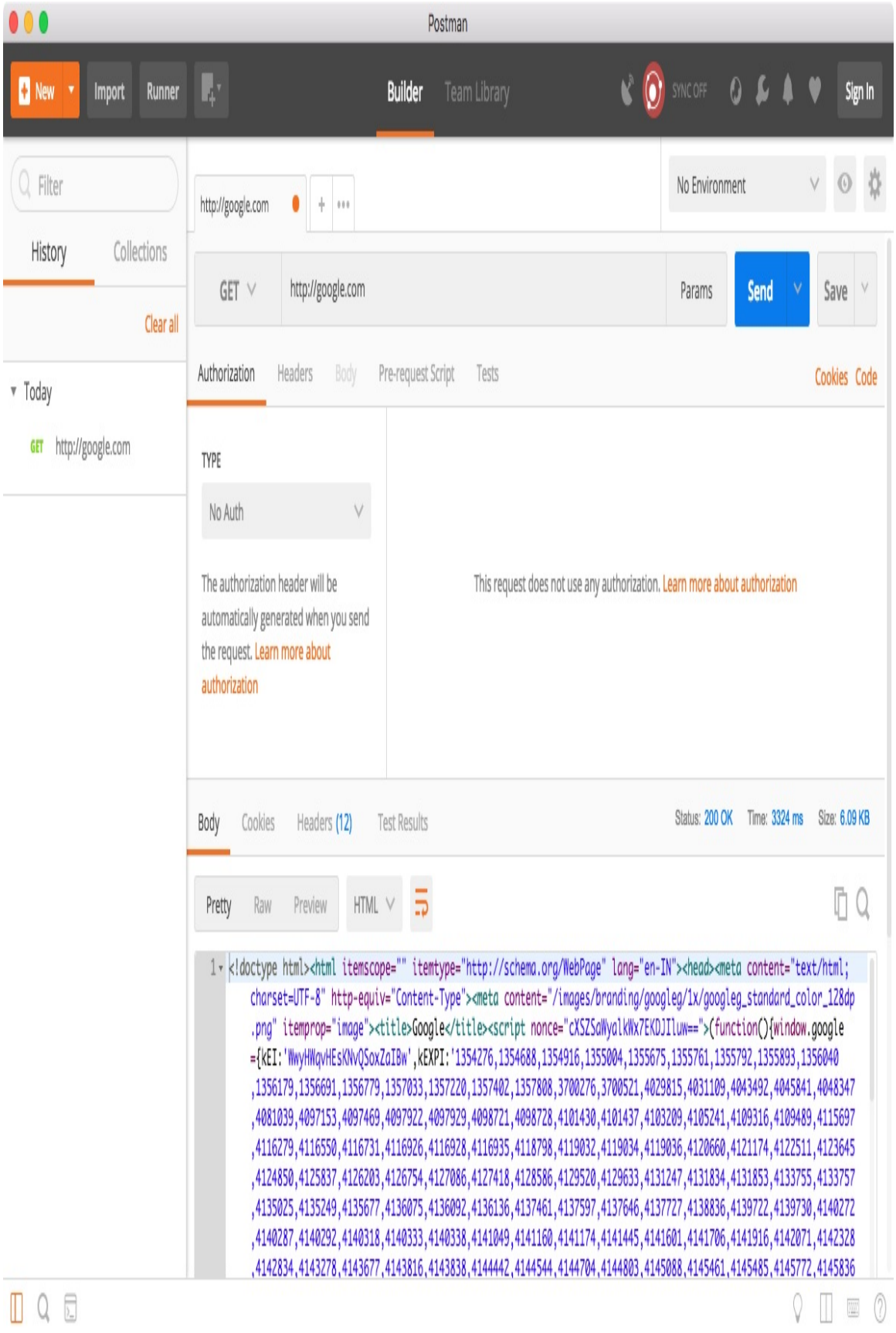
Here, we can set up our request; this is what happens in the panel:


Postman
New
Import
Runner
Builder
Team Library
SYNC OFF
Sign In
Filter
History
Collections
Nothing in your history yet. Requests that you send through Postman are automatically saved here.
New Tab
No Environment
GET
Enter request URL
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
TYPE
Inherit auth from parent
The authorization header will be automatically generated when you send the request. Learn more about authorization
This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.
Response

And, in the white space, we'll be able to view the result. Let's go ahead and make a request to Google.

Making an HTTP request to Google

In the URL bar, I'm going to type `http://google.com`. We can click Send to send off that request. Make sure you have GET chosen as your HTTP method. When I fire off the request, it comes back, and all of the data that comes back is shown in the white space:

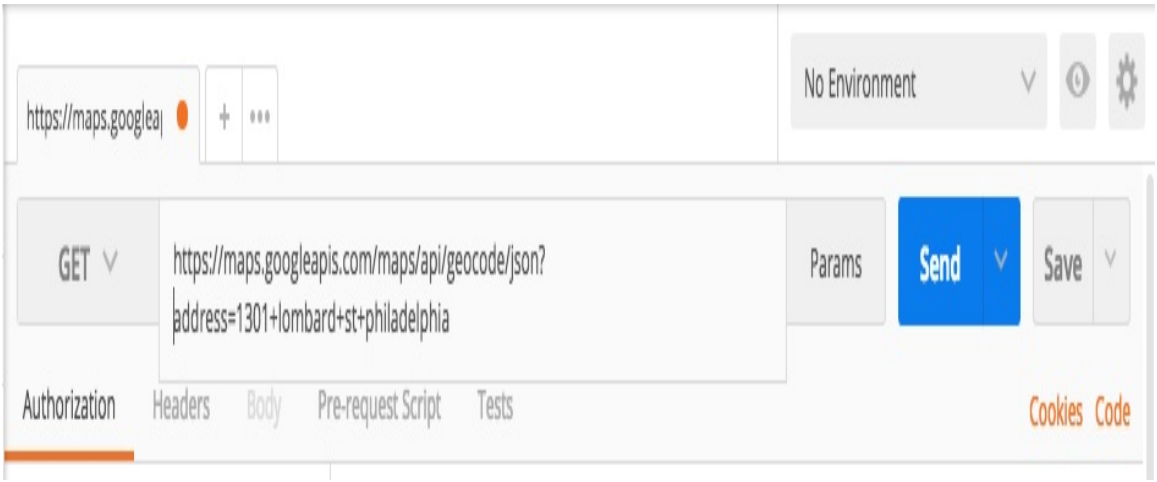

Postman
New
Import
Runner
Builder
Team Library
SYNC OFF
Sign In
Filter
History
Collections
Clear all
Today
GET http://google.com
http://google.com
No Environment
GET
http://google.com
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
TYPE
No Auth
The authorization header will be automatically generated when you send the request. Learn more about authorization
This request does not use any authorization. Learn more about authorization
Body
Cookies
Headers (12)
Test Results
Status: 200 OK
Time: 3324 ms
Size: 6.09 KB
Pretty
Raw
Preview
HTML
1
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-IN"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itemprop="image"><title>Google</title><script nonce="cXSZSaWyalkWx7EKDJIluw==">(function(){window.google={kEI:'WwyHWqvHEsKNvQSoxZaIBw',kEXPI:'1354276,1354688,1354916,1355004,1355675,1355761,1355792,1355893,1356040,1356179,1356691,1356779,1357033,1357220,1357402,1357808,3700276,3700521,4029815,4031109,4043492,4045841,4048347,4081039,4097153,4097469,4097922,4097929,4098721,4098728,4101430,4101437,4103209,4105241,4109316,4109489,4115697,4116279,4116550,4116731,4116926,4116928,4116935,4118798,4119032,4119034,4119036,4120660,4121174,4122511,4123645,4124850,4125837,4126203,4126754,4127086,4127418,4128586,4129520,4129633,4131247,4131834,4131853,4133755,4133757,4135025,4135249,4135677,4136075,4136092,4136136,4137461,4137597,4137646,4137727,4138836,4139722,4139730,4140272,4140287,4140292,4140318,4140333,4140338,4141049,4141160,4141174,4141445,4141601,4141706,4141916,4142071,4142328,4142834,4143278,4143677,4143816,4143838,4144442,4144544,4144704,4144803,4145088,4145461,4145485,4145772,4145836

We have things like the Status code; we have a 200, meaning things went great; we have the Time, which took about a quarter of a second; we have Headers, which are coming back from Google; we have Cookies, but there's none in this case; and we have our Body data. The body for `google.com` is an HTML website. For the most part, the bodies that we'll be sending and getting in Postman are going to be JSON since we're building out the REST API.

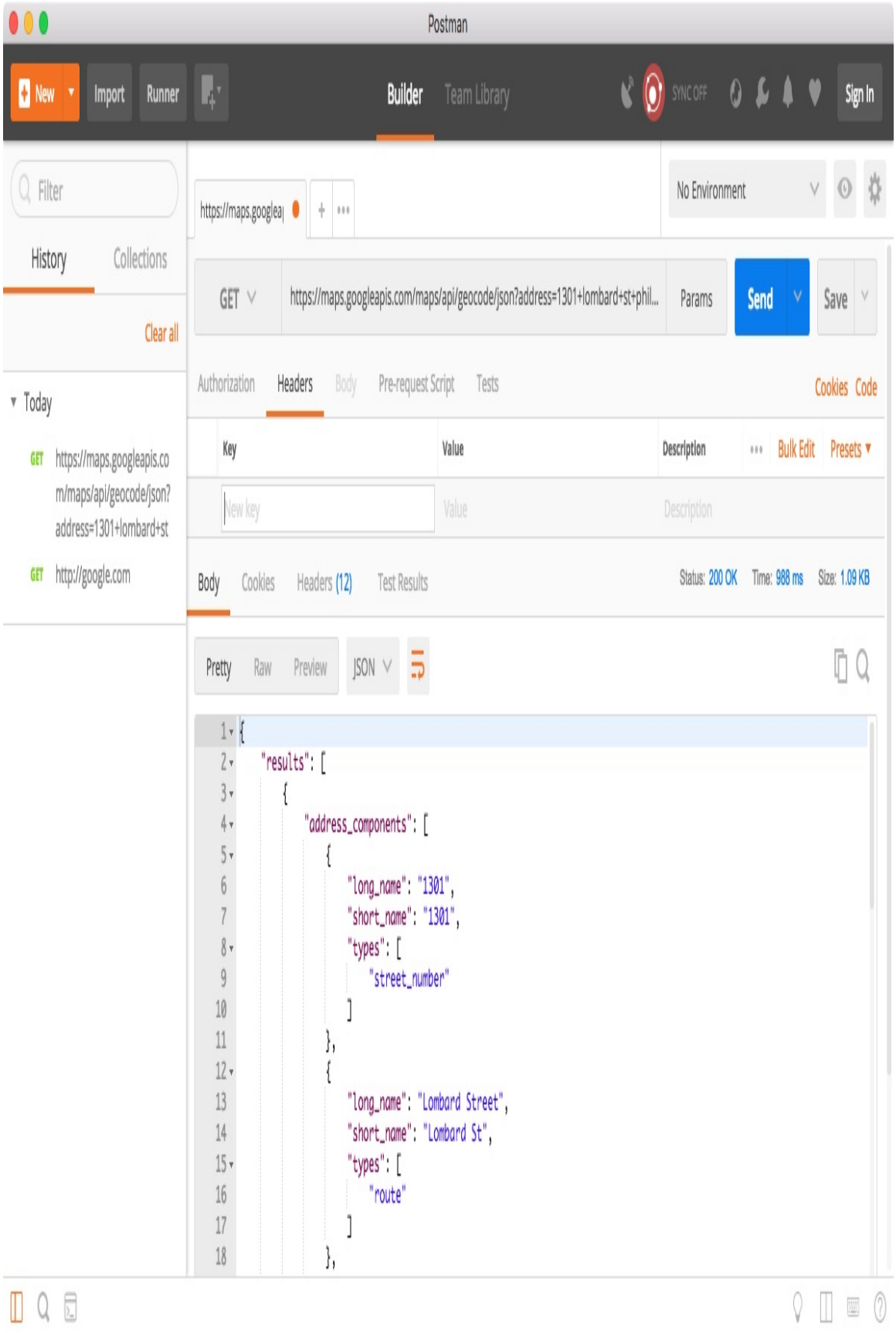
Illustrating working of the JSON data

So to illustrate how JSON data works, we're going to make a request to the geocoding URL that we used earlier in the course. If you remember, we were able to pass in a location and we got some JSON back, describing things like the latitude and longitude, and the formatted address. Now this should still be in your Chrome history.

If you deleted your history, you can go ahead and put `https://maps.googleapis.com/maps/api/geocode/json?address=1301+lombard+st+philadelphia` in the address bar. This is the URL I'll be using; you can simply copy it, or you can grab any JSON API URL. I'm going to copy it to the clipboard, head back into Postman, and swap out the URL with the URL I just copied:



Now, I can go ahead and fire off the request. We get our JSON data, which is fantastic:


Postman
New
Import
Runner
Builder
Team Library
SYNC OFF
Sign In
Filter
History
Collections
Clear all
Today
GET https://maps.googleapis.com/maps/api/geocode/json?address=1301+lombard+st
GET http://google.com
https://maps.googlea
No Environment
GET
https://maps.googleapis.com/maps/api/geocode/json?address=1301+lombard+st+phil...
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
Key
Value
Description
Bulk Edit
Presets
New key
Value
Description
Body
Cookies
Headers (12)
Test Results
Status: 200 OK
Time: 988 ms
Size: 1.09 KB
Pretty
Raw
Preview
JSON
{
"results": [
{
"address_components": [
{
"long_name": "1301",
"short_name": "1301",
"types": [
"street_number"
]
},
{
"long_name": "Lombard Street",
"short_name": "Lombard St",
"types": [
"route"
]
},

We're able to see exactly what comes back when we make this request, and this is how we're going to be using Postman.

We'll use Postman to make requests, add Todos, delete Todos, get all of our Todos, and log in; all of that stuff is going to happen right in here. Remember, APIs don't necessarily have a frontend. Maybe it's an Android app; maybe it's an iPhone app or a web app; maybe it's another server. Postman gives us a way to interact with our API, making sure it works as expected. We have all of the JSON data that comes back. In the Raw view, which is under Body, we have the raw data response. Essentially, it's just unprettified; there is no formatting, there is no colorization. We also have a Preview tab. The Preview tab is pretty useless for JSON. When it comes to JSON data., I always stick with the Pretty tab, which should be the default.

Now that we have Postman installed and we know a little bit about how to use it, we're going to move on to the next section, where we will actually create our first request. We'll be firing off a Postman request to hit the URL we're going to create. This is going to let us make new Todos right from Postman or any other application, whether it's a web app, a mobile app, or another server. That's all coming up next, so just make sure you have Postman installed. If you were able to do everything in this section, you are ready to continue.

Resource Creation Endpoint - POST /todos

In this section, you're going to create your `HTTP POST` route for adding new Todos. Before we dive into that, we're first going to refactor everything we have in `server.js`. We have database configuration stuff which should live somewhere else and we have our models, which should also live in separate files. The only thing we want in `server.js` is our Express route handlers.

Refactoring the server.js file to create POST todos route

To get started, inside of the `server` folder, we're going to make a new folder called `db`, and inside of the `db` folder we'll make a file where all of this Mongoose configuration will happen. I'm going to call that file `mongoose.js`, and all we need to do is take our Mongoose configuration code right here:

```
var mongoose = require('mongoose');
mongoose.Promise = global.Promise;
mongoose.connect('mongodb://localhost:27017/TodoApp');
```

Cut it out, and to move it over into `mongoose.js`. Now, we need to export something. What we're going to export is the `mongoose` variable. So essentially, when someone requires the mongoose.js file, they're going to have Mongoose configured and they're going to get it — they're going to get back the `mongoose` variable that comes from the library. I'm going to set `module.exports` equal to an object, and on that object we'll set `mongoose` equal to `mongoose`:

```
mongoose.connect('mongodb://localhost:27017/TodoApp');

module.exports = {
  mongoose: mongoose
};
```

Now as we know, in ES6, this can be simplified. If you have a property and a variable with the same name you can shorten it, and we can take things a step further and put it all on one line:

```
module.exports = {mongoose};
```

Now we have the Mongoose configuration in a separate file, and that file can be required in the `server.js` file. I'm going to pull off the mongoose property using ES6 destructuring. Essentially, we're creating a local variable called `mongoose` equal to the mongoose property on the object, and that object is going to be the return result from requiring the file we just created. It's in the `db` directory and it's called `mongoose.js`, and we can leave off that extension:

```
var mongoose = require('./db/mongoose');
```

Now that Mongoose lives in its own place, let's do the same thing for `Todo` and `User`. This is going to happen in a new folder in a server called `models`.

Configuring the Todo and Users file

Inside of `models`, we're going to create two files, one for each model. I'm going to make two new files called `todo.js`, and `user.js`. We can take the todos and Users models from the `server.js` file and simply copy and paste them into their appropriate files. Once the model's copied, we can remove it from `server.js`. The Todos model is going to look like this:

```
var Todo = mongoose.model('Todo', {
  text: {
    type: String,
    required: true,
    minlength: 1,
    trim: true
  },
  completed: {
    type: Boolean,
    default: false
  },
  completedAt: {
    type: Number,
    default: null
  }
});
```

The `user.js` model is going to look like this.

```
var User = mongoose.model('User', {
  email: {
    type: String,
    required: true,
    trim: true,
    minlength: 1
  }
```

```
});
```

I'm also going to remove everything we have so far, since those examples in server.js aren't necessary anymore. We can simply leave our mongoose import statement up at the top.

Inside of these model files, there are a few things we need to do. First up, we will call the `mongoose.model` in both Todos and Users files, so we still need to load in Mongoose. Now, we don't have to load in the `mongoose.js` file we created; we can load in the plain old library. Let's make a variable. We'll call that variable `mongoose`, and we're going to `require('mongoose')`:

```
var mongoose = require('mongoose');

var Todo = mongoose.model('Todo', {
```

The last thing that we need to do is export the model, otherwise we can't use it in files that require this one. I'm going to set `module.exports` equal to an object, and we'll set the `Todo` property equal to the `Todo` variable; this is exactly what we did over in `mongoose.js`:

```
module.exports = {Todo};
```

And we're going to do the exact same thing in `user.js`. Inside of `user.js`, up at the top, we'll create a variable called `mongoose` requiring `mongoose`, and at the bottom we'll export the `User` model, `module.exports`, setting it equal to an object where `User` equals `User`:

```
Var mongoose = require('mongoose');

var User = mongoose.model('User', {
  email: {
    type: String,
```

```
    required: true,
    trim: true,
    minlength: 1
  }
});

module.exports = {User};
```

Now, all three of our files have been formatted. We have our three new files and our one old one. The last thing left to do is load in `Todo` and `User`.

Loading Todo and User file in server.js

In the `server.js` file, let's make a variable using destructuring call `Todo`, setting it equal to `require('./models/todo')`, and we can do the exact same thing for `User`. Using ES6 destructuring, we're going to pull off that `User` variable, and we're going to get it from the object that comes back from a call to `require`, requiring `models/user`:

```
var {mongoose} = require('./db/mongoose');
var {Todo} = require('./models/todo');
var {User} = require('./models/user');
```

With this in place, we are now ready to get going. We have the exact same setup, only it's been refactored, and this is going to make it a lot easier to test, update, and manage. The `server.js` file is just going to be responsible for our routes.

Configuring the Express application

Now, to get started, we're going to need to install Express. We've already done that in the past, so over in the Terminal all we need to do is run `npm i` followed by the module name, which is `express`. We'll be using the most recent version, `4.16.2`.

We're also going to be installing a second module, and we can actually type that right after the first one. There's no need to run `npm install` twice. This one is called the `body-parser`. The `body-parser` is going to let us send JSON to the server. The server can then take that JSON and do something with it. `body-parser` essentially parses the body. It takes that string body and turns it into a JavaScript object. Now, with `body-parser`, we're going to be installing version `1.18.2`, the most recent version. I'm also going to provide the `--save` flag, which is going to add both Express and `body-parser` to the dependencies section of `package.json`:

```
npm i express@4.16.2 body-parser@1.18.2 --save
```

Now, I can go ahead and fire off this request, installing both modules, and over inside of `server.js`, we can start configuring our app.

First up, we have to load in those two modules we just installed. As I mentioned previously, I like to keep a space between local imports and library imports. I'm going to use a variable called `express` to store the Express library, namely `require('express')`. We're going to do the same thing for `body-parser` with a variable called `bodyParser`, setting it equal to the return result from requiring `body-parser`:

```
var express = require('express');
var bodyParser = require('body-parser');

var {mongoose} = require('./db/mongoose');
var {Todo} = require('./models/todo');
var {User} = require('./models/user');
```

Now that we can set up a very basic application. We're going to make a variable called `app`; this is going to store our Express application. I'm going to set this equal to a call to `express`:

```
var {User} = require('./models/user');

var app = express();
```

And we're also going to call `app.listen`, listening on a port. We will be deploying this to Heroku eventually. For now though, we're going to have a local port, port `3000`, and we'll provide a callback function that's going to fire once the app is up. All we're going to do is use `console.log` to print `Started on port 3000`:

```
var app = express();

app.listen(3000, () => {
  console.log('Started on port 3000');
});
```

Configuring the POST route

Now, we have a very basic server. All we have to do is start configuring our routes, and as I promised, the one we're going to be focusing on in this section is the POST route. This is going to let us create new Todos. Now, inside of your REST APIs, there's the basic CRUD operations, CRUD being Create, Read, Update, and Delete.

When you want to create a resource, you use the `POST HTTP` method, and you send that resource as the body. This means that when we want to make a new Todo, we're going to send a JSON object over to the server. It's going to have a `text` property, and the server is going to get that `text` property, create the new model, and send the complete model with the ID, the completed property, and `completedAt` back to the client.

To set up a route, we need to call `app.post`, passing in the two arguments we've used for every single Express route, which are our URL and our callback function that get called with the `req` and `res` objects. Now, the URL for a REST API is really important, and there is a lot of talk about the proper structure. For resources, what I like to do is use `/todos`:

```
app.post('/todos', (req, res) => {

});
```

This is for resource creation, and this is a pretty standard setup. `/todos` is for creating a new Todo. Later on, when we want to read Todos, we'll use the `GET` method, and we will use `GET` from `/todos` to get all Todos or `/todos`, some crazy number, to get an individual Todo by its ID. This is a very common pattern, and it's the one we're going to

be using. For now though, we can focus on getting the body data that was sent from the client.

Getting body data from the client

To do this, we have to use the `body-parser` module. As I mentioned previously, `body-parser` is going to take your JSON and convert it into an object, attaching it onto this `request` object. We're going to configure the middleware using `app.use`. The `app.use` takes the middleware. If we're writing custom middleware, it'll be a function; if we're using third-party middleware, we usually just access something off of the library. In this case, it's going to be `bodyParser.json` getting called as a function. The return value from this JSON method is a function, and that is the middleware that we need to give to Express:

```
var app = express();

app.use(bodyParser.json());
```

With this in place, we can now send JSON to our Express application. What I'd like to do inside of the `post` callback is simply `console.log` the value of `req.body`, where the body gets stored by `bodyParser`:

```
app.use(bodyParser.json());

app.post('/todos', (req, res) => {
  console.log(req.body);
});
```

We can now start up the server and test things out inside of Postman.

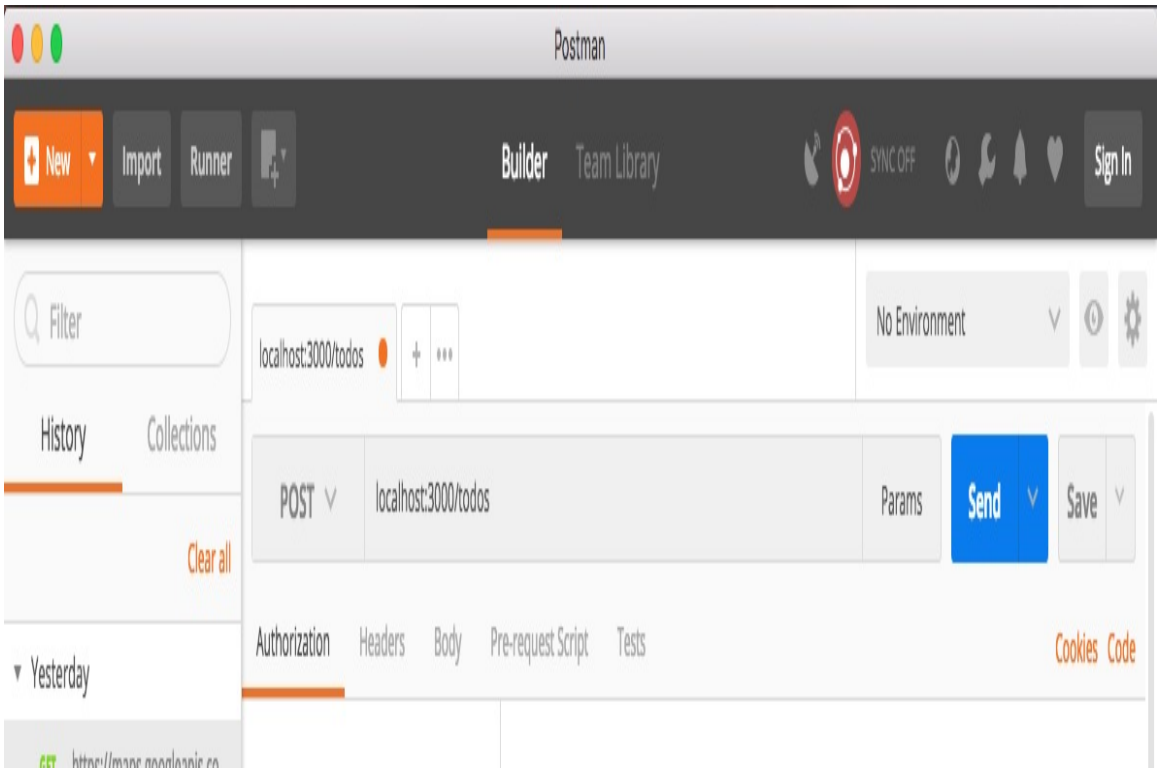
Testing the POST route inside Postman

In the Terminal, I'm going to use `clear` to clear the Terminal output, and then I'll run the app:

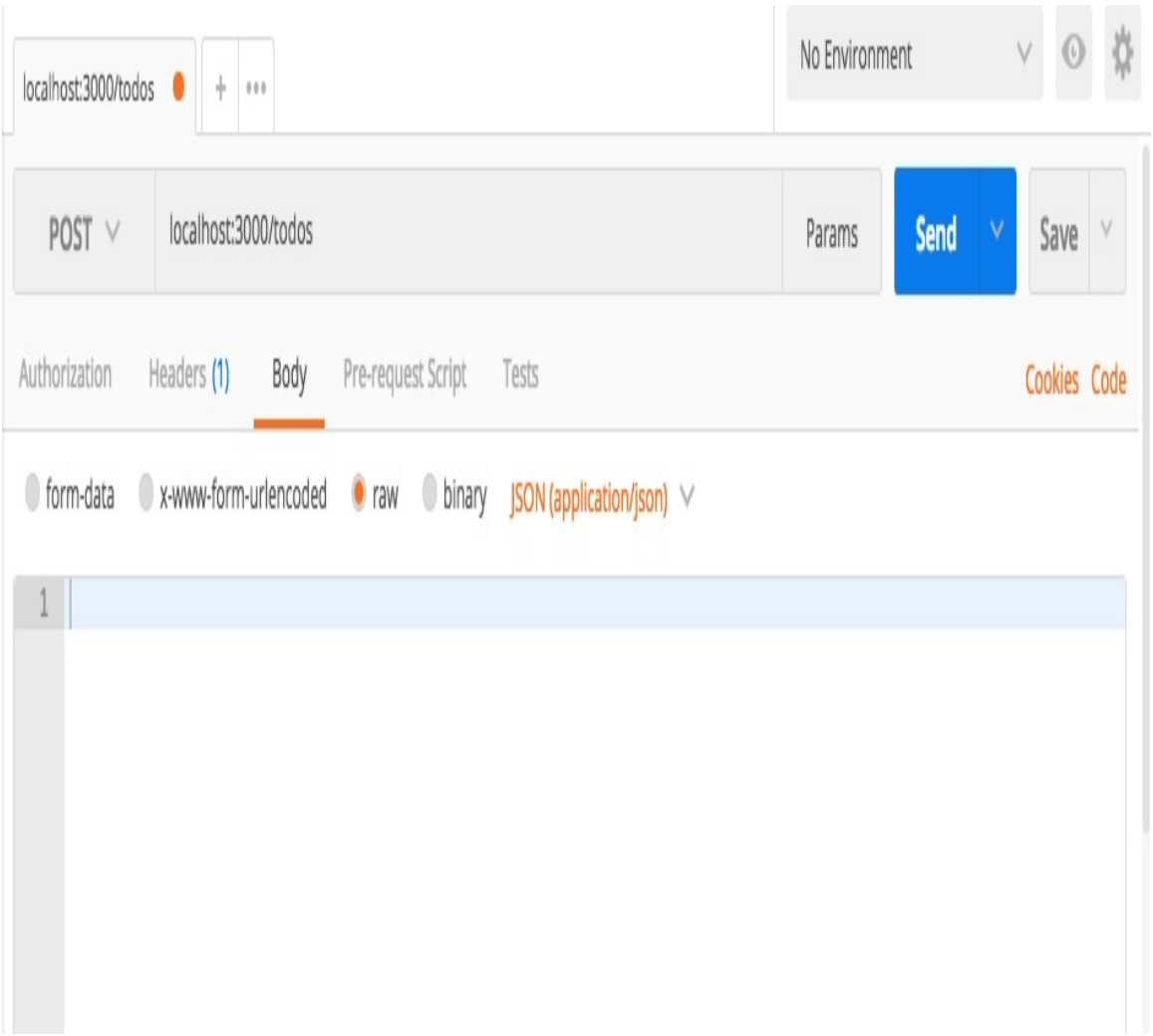
```
node server/server.js
```

The server is up on port 3000, which means we can now head into Postman:

Inside of Postman, we're not going to be making a `GET` request like we did in the previous section. This time, what we're going to be doing is making a POST request, which means we need to change the HTTP method to POST, and type the URL. That's going to be `localhost:3000` for the port, `/todos`. This is the URL that we want to send our data to:



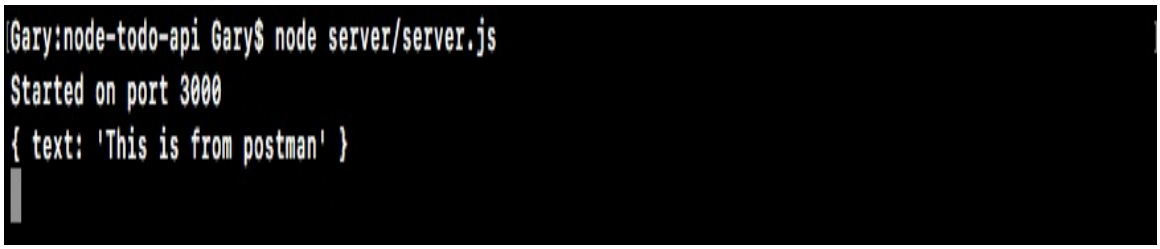
Now in order to send some data to the application, we have to go to the Body tab. We're trying to send JSON data, so we're going to go to raw and select JSON (application/json) from the drop-down list on the right-hand side:



Now we have our Header set. This is the Content-Type header, letting the server know that JSON is getting sent. All of this is done automatically with Postman. Inside of Body, the only piece of information I'm going to attach to my JSON is a `text` property:

```
{
  "text": "This is from postman"
}
```

Now we can click Send to fire off our request. We're never going to get a response because we haven't responded to it inside of `server.js`, but if I head over to the Terminal, you see we have our data:



This is the data we created inside of Postman. It's now showing up in our Node application, which is fantastic. We are one step closer to actually creating that Todo. The only thing left to do inside of the post handler is to actually create the Todo using the information that comes from the `User`.

Creating an instance of Mongoose model

Inside `server.js`, let's make a variable called `todo` to do what we've done previously, creating an instance of a Mongoose model. We're going to set it equal to `new Todo`, passing in our object and passing in the values we want to set. In this case, we just want to set `text`. We're going to set text to `req.body`, which is the object we have, and then we're going to access the `text` property, like so:

```
app.post('/todos', (req, res) => {
  var todo = new Todo({
    text: req.body.text
  });
```

Next up, we're going to call `todo.save`. This is going to actually save the model to the database, and we're going to be providing a callback for a success case and an error case.

```
app.post('/todos', (req, res) => {
  var todo = new Todo({
    text: req.body.text
  });

todo.save().then((doc) => {

}, (e) => {

});
```

Now if things go well, we're going to be sending back the actual Todo which is going to show up in the then callback. I'm going to get the `doc`, and right inside of the callback function, I'm going to use

`res.send` to send the doc back. This is going to give the `User` really important information, things like the ID and the `completed` and `completedAt` properties, which were not set by the `User`. If things go poorly and we get an error, that's fine too. All we're going to do is use `res.send` to send that error back:

```
todo.save().then((doc) => {
  res.send(doc);
}, (e) => {
  res.send(e);
});
```

We'll be modifying how we send errors back a little later. For now, this code is going to work just great. We can also set an HTTP status.

Setting up HTTP status code

If you remember, HTTP statuses let you give someone some information about how the request went. Did it go well? Did it go poorly? That kind of thing. You can get a list of all the HTTP statuses available to you by going to `httpstatuses.com`. Here, you can view all of the statuses that you can set:

HTTP Status Codes

httpstatuses.com is an easy to reference database of HTTP Status Codes with their definitions and helpful code references all in one place. Visit an individual status code via `httpstatuses.com/code` or browse the list below.

1xx Informational

100 Continue

101 Switching Protocols

102 Processing

2xx Success

200 OK

201 Created

202 Accepted

203 Non-authoritative Information

204 No Content

The one that's set by default by Express is `200`. This means that things went OK. What we're going to be using for an error is code `400`. A `400` status means there was some bad input, which is going to be the case if the model can't be saved. Maybe the `User` didn't provide a `text` property, or maybe the text string was empty. Either way, we want to send a `400` back, and that's going to happen. Right before we call `send`, all we're going to do is call `status`, passing in the status of `400`:

```
todo.save().then((doc) => {
  res.send(doc);
}, (e) => {
  res.status(400).send(e);
});
```

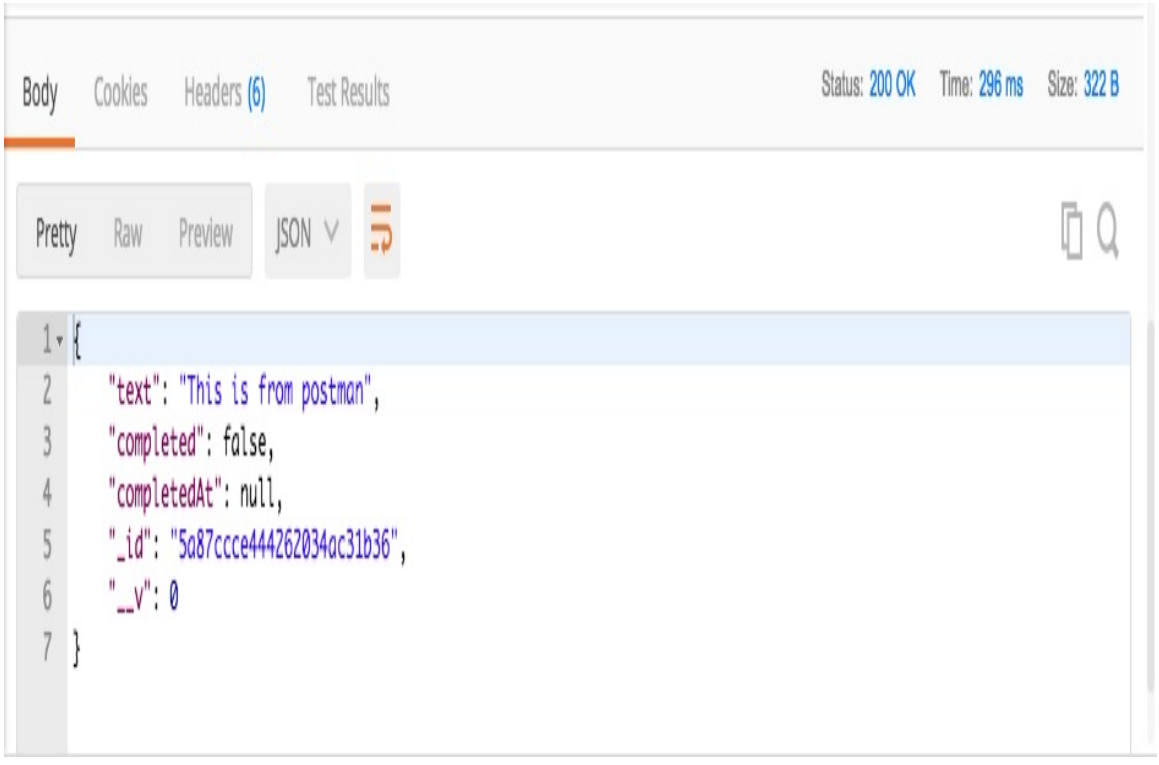
With this in place, we are now ready to test out our `POST /todos` request over inside of Postman.

Testing POST /todos inside of Postman

I'm going to restart the server in the Terminal. You could start this up with `nodemon` if you like. For the moment, I'll be manually restarting it:

```
nodemon server/server.js
```

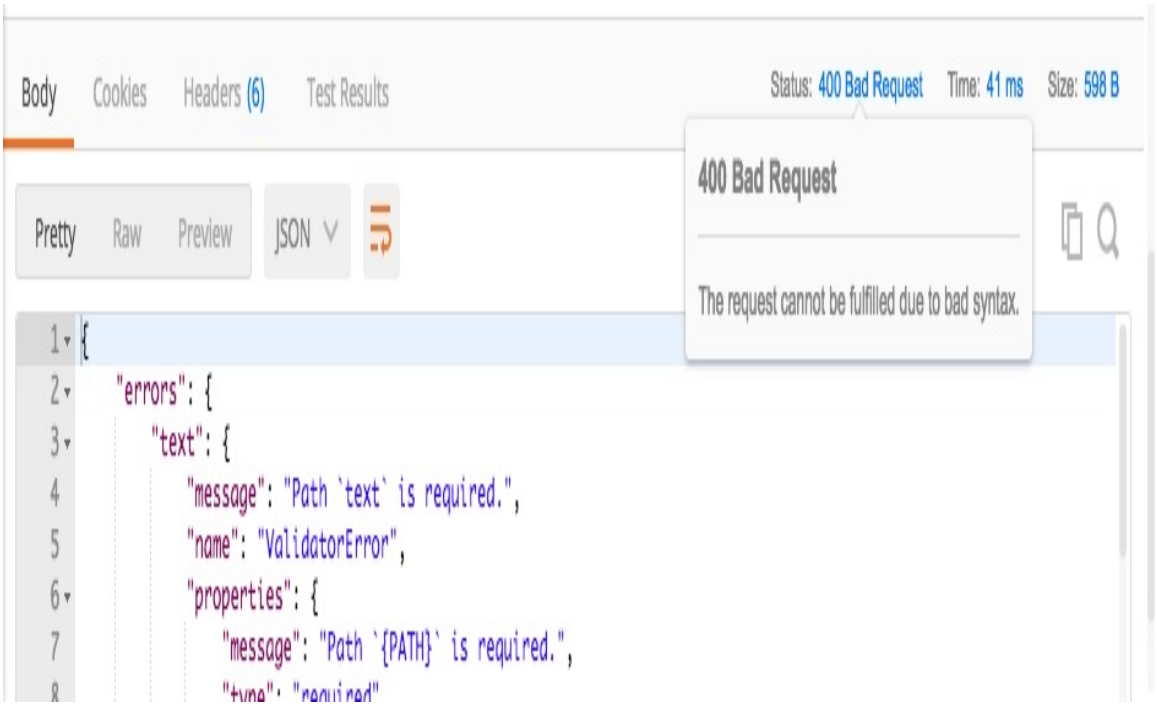
We're now up on localhost 3000, and inside of Postman, we can make the exact same request we made earlier. I'm going to click on Send:



We get a Status of 200. This is fantastic; it's the default status,

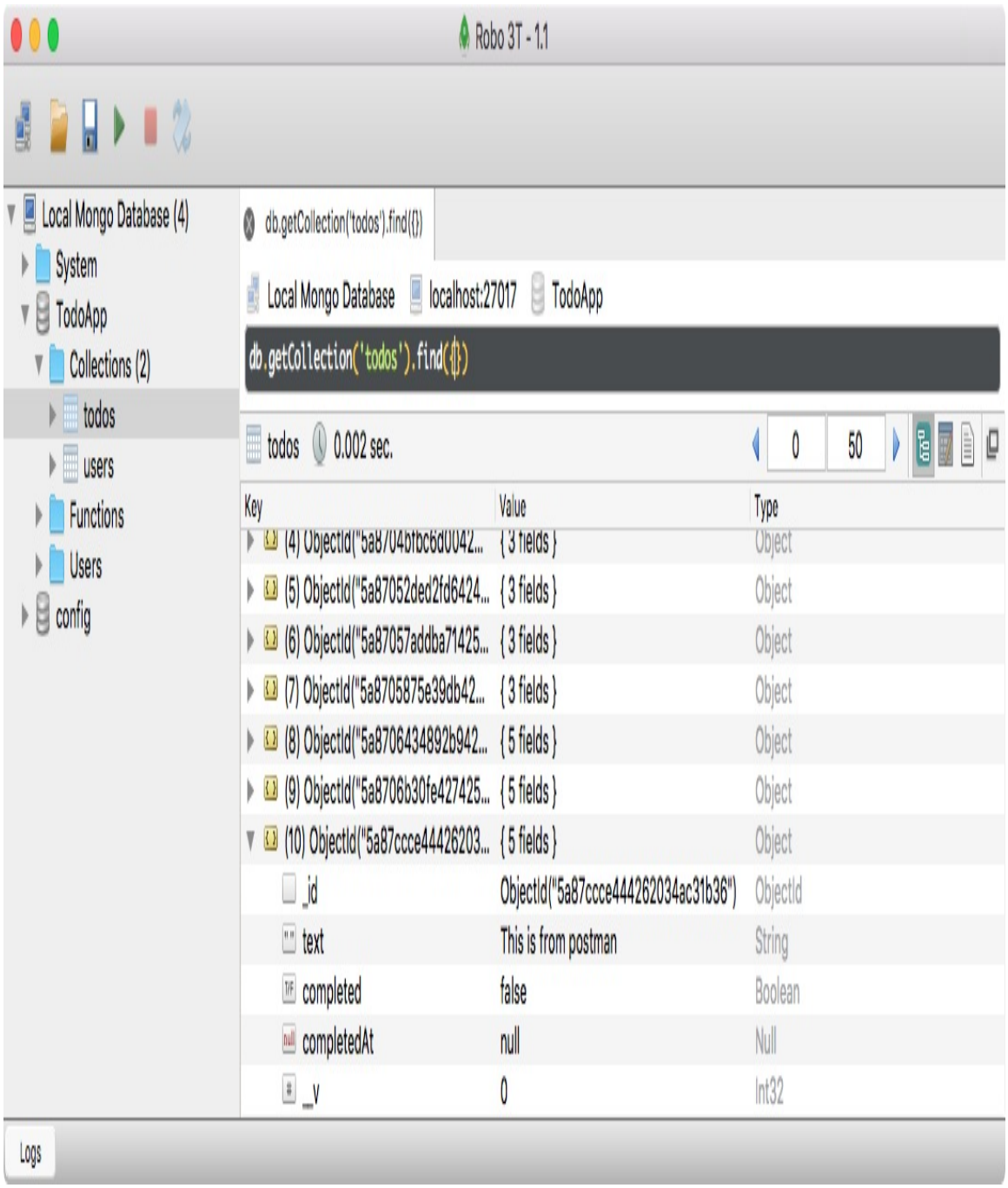
which means things went great. The JSON response is exactly what we expected. We have our `text` that we set; we have the `_id` property which was generated; we have `completedAt`, which is set to `null`, the default; and we have `completed` set to `false`, the default.

We could also test what happens when we try to create a Todo without the proper information. For example, maybe I set a `text` property equal to an empty string. If I send this over, we now get a 400 Bad Request:



Now, we have a bunch of validation code saying that the `Todo validation failed`. Then, we can go into the `errors` object to get the specific error. Here, we can see the `text` field failed, and the `message` is `Path 'text' is required`. All of this information can help someone fix their request and make a proper one.

Now if I head over into Robomongo, I'm going to refresh the collection for `todos`. Look at the last one, and it is indeed the one we created in Postman:



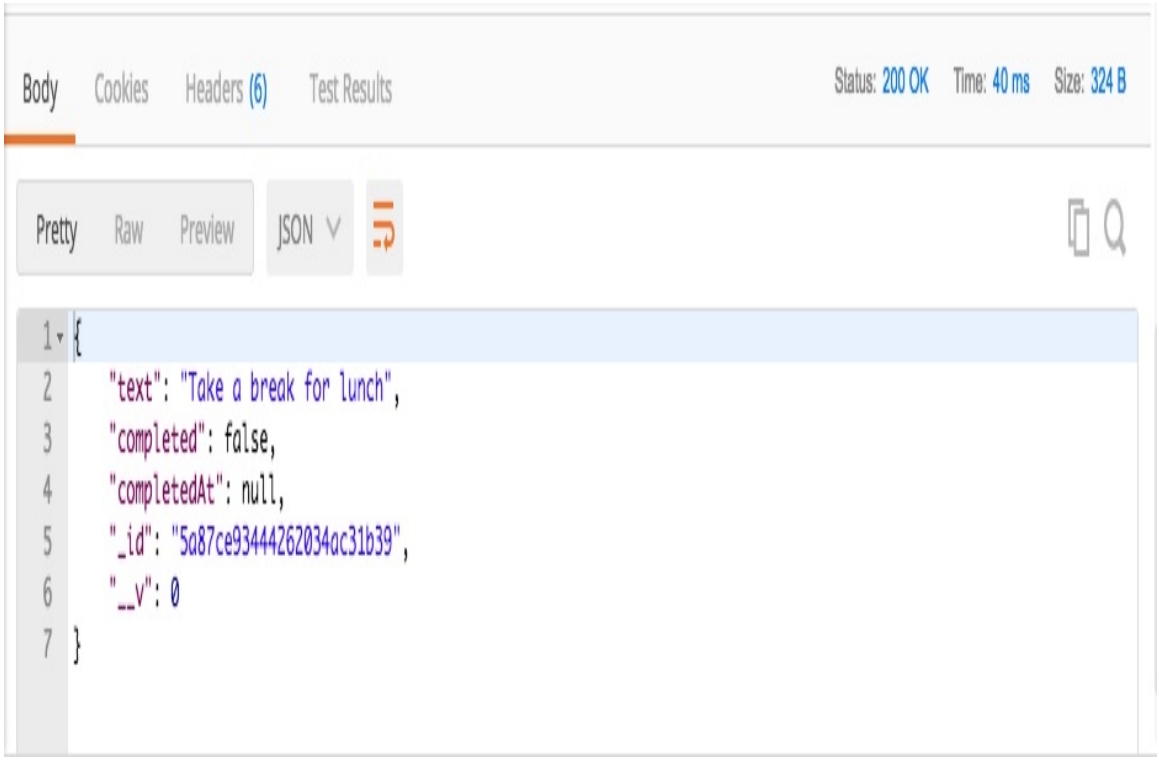
The text is equal to This is from postman. With this in place, we now have our very first HTTP endpoint set up for the Todo REST API.

Now I haven't talked exactly about what REST is. We're going to talk about that later. For now, we're going to focus on creating these endpoints. The REST version will come up a little later when we

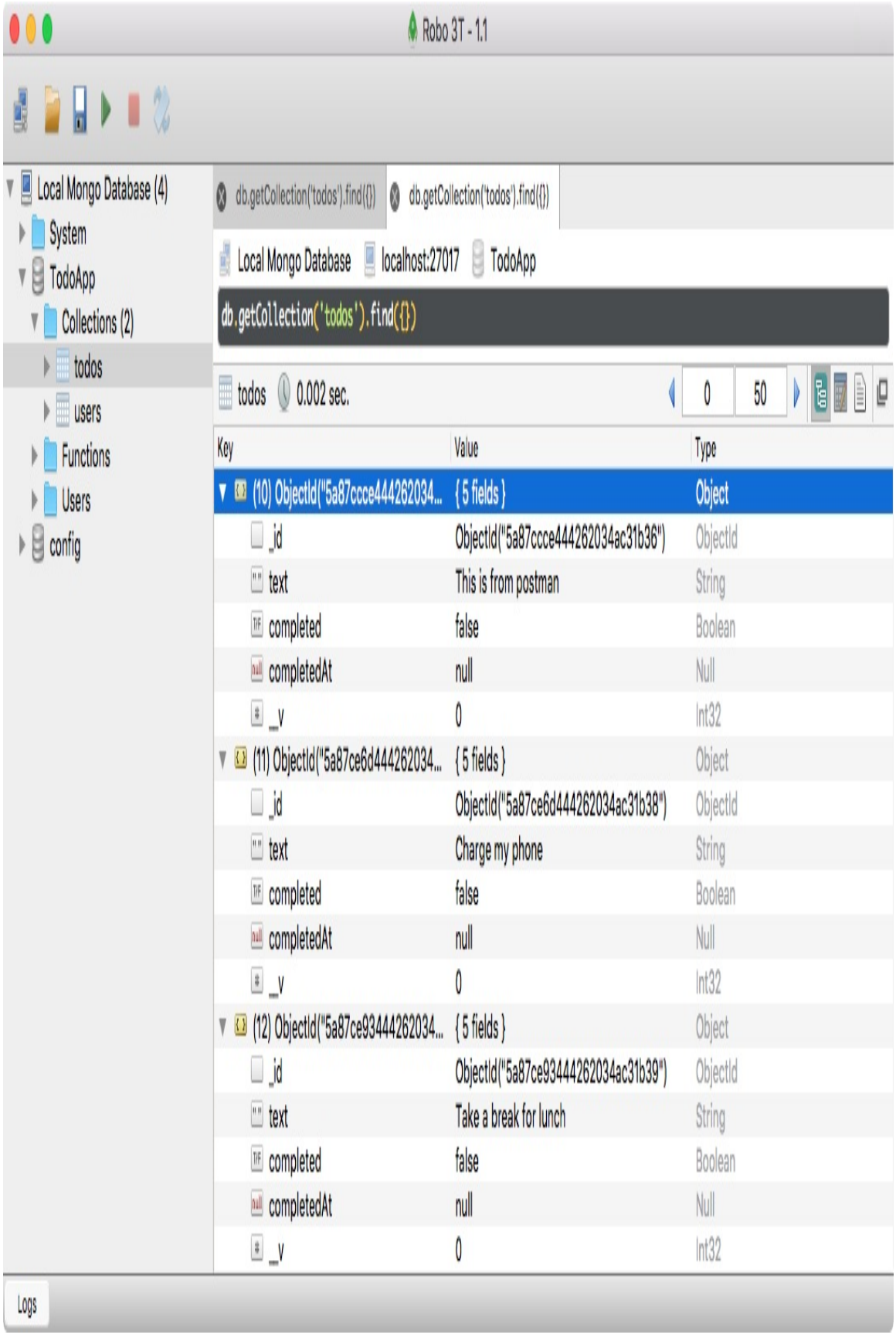
start adding authentication.

Adding more Todos to the database

Over inside of Postman, we can add a few more Todos, which is what I'm going to do. `Charge my phone`—I don't think I've ever needed to be reminded of that one—and we'll add `Take a break for lunch`. In the Pretty section, we see the `Charge my phone` Todo was created with a unique ID. I'm going to send off the second one, and we'll see that the `Take a break for lunch` Todo was created:



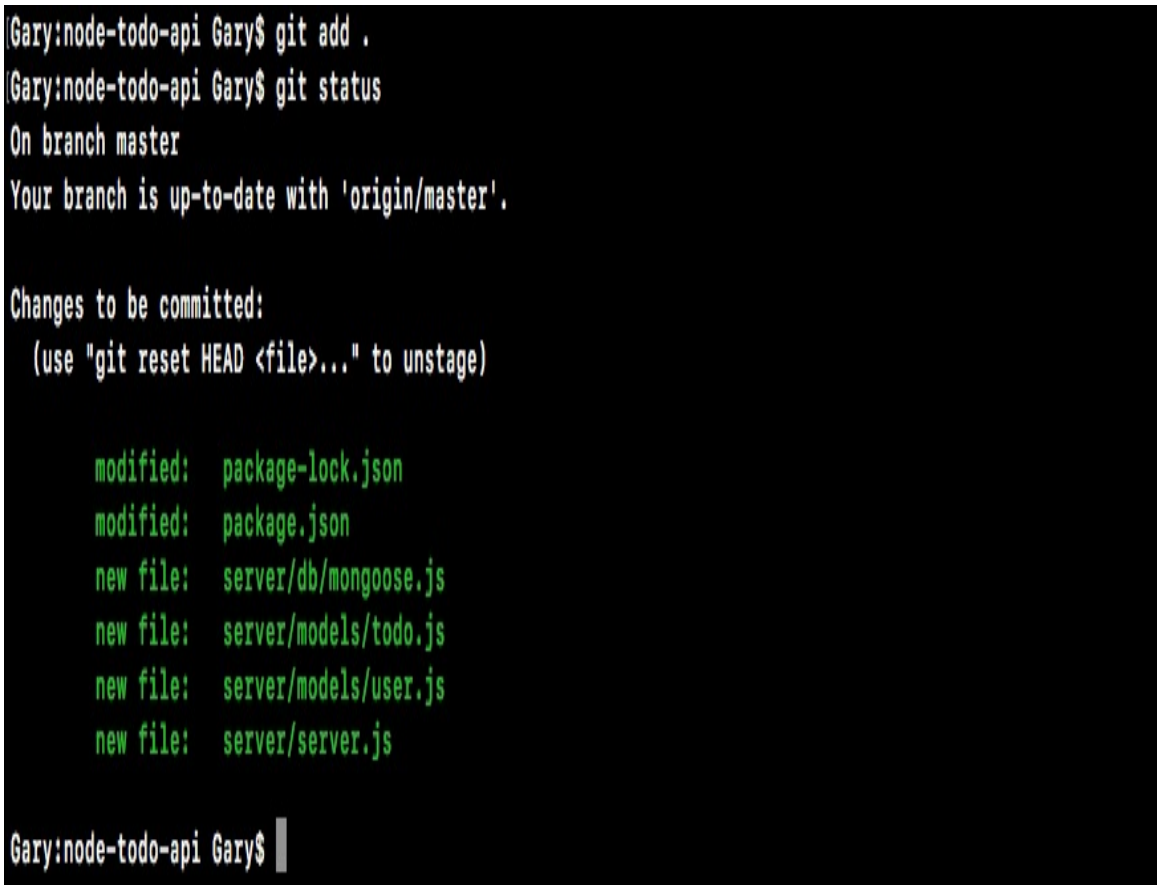
Over inside of Robomongo, we can give our `todos` collection a final refresh. I'm going to expand those last three items, and they are indeed the three items we created in Postman:


Robo 3T - 1.1
Local Mongo Database (4)
System
TodoApp
Collections (2)
todos
users
Functions
Users
config
db.getCollection('todos').find({})
db.getCollection('todos').find({})
Local Mongo Database
localhost:27017
TodoApp
db.getCollection('todos').find({})
todos
0.002 sec.
0
50
Key
Value
Type
(10) ObjectId("5a87ccce444262034...
{ 5 fields }
Object
_id
ObjectId("5a87ccce444262034ac31b36")
ObjectId
text
This is from postman
String
completed
false
Boolean
completedAt
null
Null
__v
0
Int32
(11) ObjectId("5a87ce6d444262034...
{ 5 fields }
Object
_id
ObjectId("5a87ce6d444262034ac31b38")
ObjectId
text
Charge my phone
String
completed
false
Boolean
completedAt
null
Null
__v
0
Int32
(12) ObjectId("5a87ce93444262034...
{ 5 fields }
Object
_id
ObjectId("5a87ce93444262034ac31b39")
ObjectId
text
Take a break for lunch
String
completed
false
Boolean
completedAt
null
Null
__v
0
Int32
Logs

Now that we have some meaningful work done in our project, let's go ahead and commit our changes. As you can see over in Atom, the `server` directory is green, meaning that it hasn't been added to Git, and the `package.json` file is orange, which means that it's been modified, even though Git is tracking it. Over in the Terminal we can shut down the server, and I always like to run `git status` to do a sanity check:

```
Gary:node-todo-api Gary$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   package-lock.json
        modified:   package.json

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        server/

no changes added to commit (use "git add" and/or "git commit -a")
Gary:node-todo-api Gary$
```

Here, everything does look as expected. I can it using `git add .` to add everything, followed by one more sanity check:



Here, we have our four new files in the `server` folder, as well as our `package.json` file.

Now, it's time to make that commit. I'm going to create a quick commit. I'm using the `-am` flag, which usually adds modified files. Since I already used add, I can simply use the `-m` flag, like we've been doing all the way through the course. A good message for this one would be something like `Add POST /todos route and refactor mongoose`:

```
git commit -m 'Add POST /todos route and refactor mongoose'
```

With the commit in place, we can now wrap things up by pushing it up to GitHub, making sure it's backed up, and making sure it's available for anyone else collaborating on the project. Remember, creating a commit alone does not get it up on GitHub; you've got to push that with another command, namely `git push`. With that in

place, it's now time to move on to the next section, where you will be testing the route you just created.

Testing POST /todos

In this section, you're going to learn how to set up the test suite for the Todo API, similar to what we did in the `test` section, and we'll be writing two test cases for `/todos`. We're going to verify that when we send the correct data as the body, we get a `200` back with the completed doc, including the ID; and, if we send bad data, we expect a `400` back with the error object.

Installing npm modules for testing POST /todos route

Now before we can do any of this, we have to install all of those modules we installed in the `test` section, `expect` for assertions, `mocha` for the entire test suite, `supertest` to test our Express routes, and `nodemon`. The `nodemon` module is going to let us create that `test-watch` script we had, so we can automatically restart the test suite. Now I know you have `nodemon` installed globally, but since we are using it inside of a `package.json` script, it's a great idea to install it locally as well.

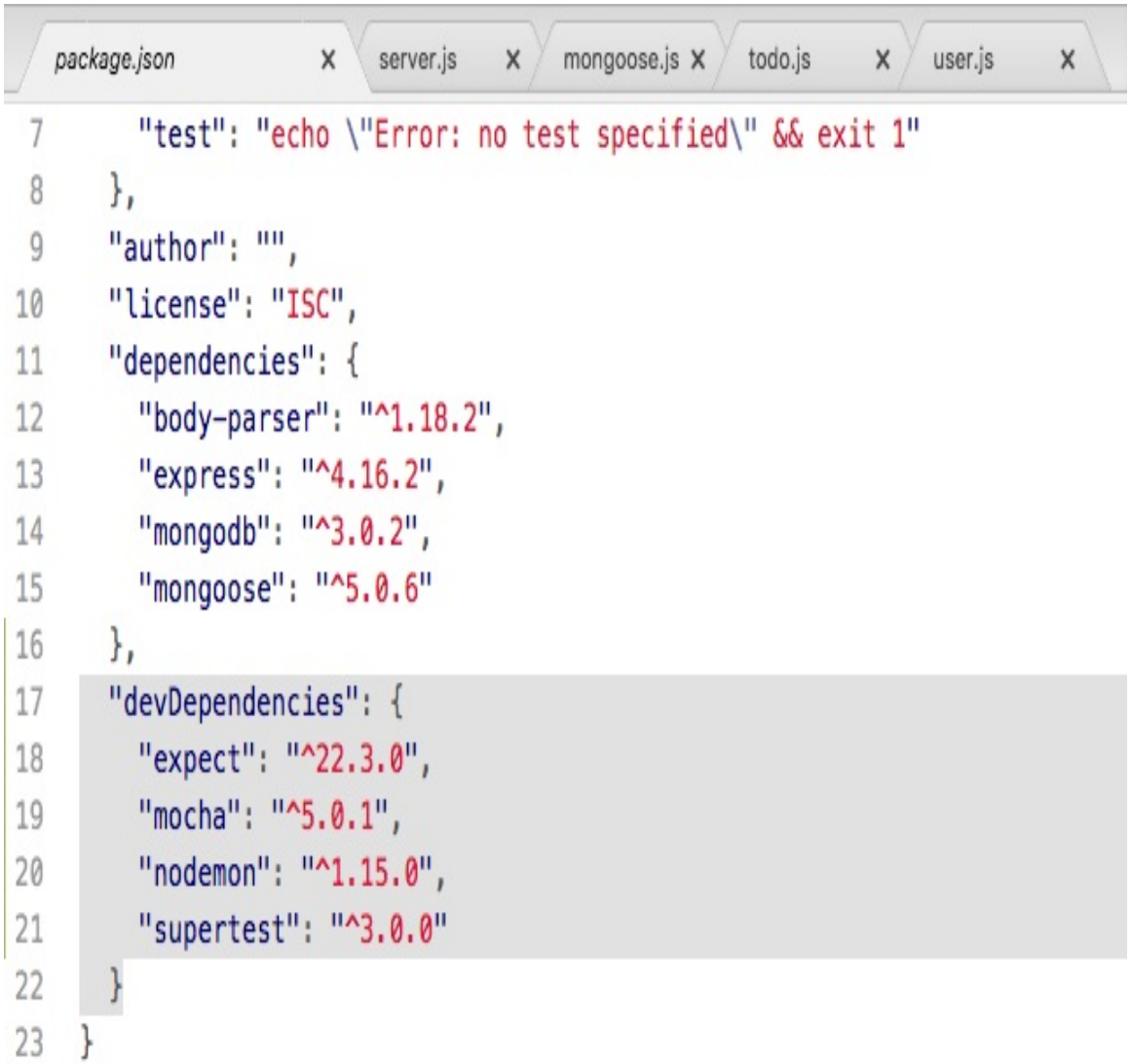
We're going to run `npm i` with `expect` version `22.3.0`, the most recent. Next up is going to be `mocha`. The most recent version is `5.0.1`. After that is `nodemon` version `1.15.0`, and last but not least is `supertest` at version `3.0.0`. With this in place, all we have to do is tack on that `--save-dev` flag. We want to save these, but not as regular dependencies. They're for testing purposes only, so we're going to save them as `devDependencies`:

```
npm i expect@22.3.0 mocha@5.0.1 nodemon@1.15.0 supertest@3.0.0 --save-dev
```

Now, we can go ahead and run this command, and once it's done we'll be able to start setting up the test files inside of Atom.

Setting up the test files

In Atom, inside my `package.json` file, I now have my `devDependencies` listed out:



Now, my output for this command might look a little different than yours. npm is caching some of my modules that I've installed recently, so as you can see in the preceding screenshot, it's just

grabbing the local copy. They did indeed get installed though, and I can prove that by opening up the `node_modules` folder.

We're now going to create a folder inside the `server` where we can store all of our test files, and this folder is going to be called `tests`. The only file we're going to worry about creating for this section is a test file for `server.js`. I'm going to make a new file in tests called `server.test.js`. This is the extension we'll be using for test files in the chapter. Inside of the `server.test` file, we can now kick things off by requiring a lot of those modules. We're going to require the `supertest` module and `expect`. The `mocha` and `nodemon` modules do not need to be required; that's not how they're used.

The `const expect` variable we'll get will be equal to `require('expect')`, and we'll do the exact same thing for `supertest`, using `const`:

```
const expect = require('expect');
const request = require('supertest');
```

Now that we have these in place, we need to load in some of our local files. We need to load in `server.js` so we have access to the Express app since we need that for super-test, and we also want to load in our Todo model. As you'll see a little later, we're going to be querying the database, and having access to this model is going to be necessary. Now the model already exports something, but `server.js` currently exports nothing. We can fix this by adding `module.exports` to the very bottom of the `server.js` file, setting it equal to an object. On that object, all we're going to do is set the `app` property equal to the `app` variable, using the ES6 object syntax.

```
module.exports = {app};
```

With this in place, we are now ready to load those two files in.

Loading the test files

First up, let's go ahead and create a local variable called `app`, and we're going to be using ES6 destructuring to pull it off of the return result from requiring the server file. Here, we're going to start by getting the relative path. Then, we're going to go back one directory from `tests` into `server`. The filename is simply `server` without the extension. We can do the exact same thing for the Todo model as well.

We're going to make a constant called `Todo`. We're using ES6 destructuring to pull that off of the export, and the file is from the relative path, back a directory. Then we have to go into the `models` directory, and finally, the filename is called `todo`:

```
const expect = require('expect');
const request = require('supertest');

const {app} = require('./../server');
const {Todo} = require('./../models/todo');
```

And now that we have all of this loaded in, we are ready to create our `describe` block and add our test cases.

Adding describe block for the test cases

I'm going to use `describe` to group all of the routes. I'm going to have multiple test cases for some routes, and it's nice to add a `describe` block so you can quickly glance at the test output in the Terminal. The `describe` block for POST Todos will simply be called `POST /todos`. Then, we can add our arrow function (`=>`), and inside of here we can start laying out our test cases. The first test is going to verify that when we send the appropriate data, everything goes as expected:

```
const {Todo} = require('./../models/todo');

describe('POST /todos', () => {
  it('should create a new todo')
});
```

Now, we can add our callback function, and this function is going to take the `done` argument because this is going to be an asynchronous test. You have to specify `done`, otherwise this test is not going to work as expected. In the callback function, what we're going to do is create a variable called `text`. This is the only setup data we really need. We just need a string, and we're going to use that string throughout. Go ahead and give this any value you like. I'm going to use `Test todo text`.

```
describe('POST /todos', () => {
  it('should create a new todo',(done) => {
    var text = 'Test todo text';
  });
});
```

Now it's time to start making that request via `supertest`. We only made `GET` requests previously, but `POST` requests are just as easy.

Making the POST requests via supertest

We're going to call request, passing in the app we want to make the request on. Next up, we're going to call `.post`, which sets up a `POST` request. We're going to go to `/todos`, and the new thing we're going to do is actually send data. In order to send data along with the request as the body we have to call `send`, and we're going to pass in an object. This object is going to get converted to JSON by `supertest`, so there's no need for us to worry about that—just another great reason to use the `supertest` library. We're going to set `text` equal to the `text` variable shown previously, and we can use the ES6 syntax to get that done:

```
describe('POST /todos', () => {
  it('should create a new todo',(done) => {
    var text = 'Test todo text';

    request(app)
    .post('/todos')
    .send({text})
  })
});
```

Now that we've sent the request, we can start making assertions about the request.

Making assertions about the POST request

We'll start with the status. I'm going to `expect` that the status equals `200`, which should be the case when we send across valid data. After this, we can go ahead and make an assertion about the body that comes back. We want to make sure the body is an object and that it has the `text` property equal to the one we specified previously. That's exactly what it should be doing when it sends the body back.

Over inside of `server.test.js`, we can get that done by creating a custom `expect` assertion. If you can recall, our custom `expect` calls do get passed in the response, and we can use that response inside of the function. We're going to `expect` that the response body has a `text` property and that the `text` property equals using `toBe`, the `text` string we have defined:

```
request(app)
.post('/todos')
.send({text})
.expect(200)
.expect((res) => {
  expect(res.body.text).toBe(text);
})
```

If that's the case, great; this will pass. If not, that's fine too. We're just going to throw an error and the test will fail. The next thing we need to do is call end to wrap things up, but we're not quite done yet. What we want to do is actually check what got stored in the MongoDB collection, and this is why we loaded in the model. Instead of passing `done` into end like we did previously, we're going to pass in a function. This function will get called with an error, if

any, and the response:

```
  request(app)
  .post('/todos')
  .send({text})
  .expect(200)
  .expect((res) => {
    expect(res.body.text).toBe(text);
  })
  .end((err, res) => {

});
```

This callback function is going to allow us to do a few things. First up, let's handle any errors that might have occurred. This will be if the status wasn't `200`, or if the `body` doesn't have a `text` property equal to the `text` property we sent in. All we have to do is check if an error exists. If an error does exist, all we're going to do is pass it into `done`. This is going to wrap up the test, printing the error to the screen, so the test will indeed fail. I'm also going to `return` this result.

```
.end((err, res) => {
  if(err) {
    return done(err);
  }
});
```

Now, returning it doesn't do anything special. All it does is stop the function execution. Now, we're going to make a request to the database fetching all the Todos, verifying that our one `Todo` was indeed added.