

# Learn the MERN stack by building an exercise tracker — MERN Tutorial

 [medium.com/@beaucarnes/learn-the-mern-stack-by-building-an-exercise-tracker-mern-tutorial-59c13c1237a1](https://medium.com/@beaucarnes/learn-the-mern-stack-by-building-an-exercise-tracker-mern-tutorial-59c13c1237a1)

June 16, 2020



Beau Carnes

Jun 25, 2019

.

20 min read

.



The MERN stack is a popular stack of technologies for building a modern single-page application. In this tutorial, you will learn the MERN stack by building an exercise tracker.

The MERN stack consists of the following technologies:

- : A document-based open source database.
- : A web application framework for Node.js.
- : A JavaScript front-end library for building user interfaces.
- : JavaScript run-time environment that executes JavaScript code outside of a browser (such as a server).

It is also common to use **Mongoose**, which is a simple, schema-based solution to model application data.

We'll be hosting our database in the cloud using MongoDB Atlas and Google Cloud Platform.

First we'll review MongoDB and create a MongoDB Atlas account, then we'll code the app.

Before we get into it, I want to let you know that this tutorial was sponsored by MongoDB.

## MongoDB Overview

Tabular (Relational)	MongoDB
Database	Database
Table	Collection
Row	Document
Index	Index
Join	\$lookup
Foreign Key	Reference

Relational vs MongoDB database

In the tabular, or relational world, we think of things like databases, tables, row, etc. MongoDB has similar concepts that use different terms.

Instead of table, we have collections. Instead of rows, we have documents. We can do JOIN operations with the \$lookup operator. And instead of foreign keys we utilize references.

MongoDB is very well suited for handling data with a wide variety of relationships. Let's have a quick look at the document model to see.

```
{
  name : "Beau Carnes",
  title : "Developer & Teacher",
  address : {
    address_1 : "123 Main Street",
    city : "Grand Rapids",
    state : "Michigan",
    postal_code : "49503"
  },
  topics : [ "MongoDB", "Python", "JavaScript", "Robots" ],
  employee_number : 1234,
  location : [44.9901, 123.0262]
}
```

String

Nested Document

Array

Integer

Geo-Spatial Coordinates

Example MongoDB document

MongoDB stores data on disk in the BSON format, or Binary JSON. This provides a wide variety of support for data types, far beyond those supported by JSON, like Decimal128, ISODate, and more.

The document model also allows for nesting documents inside each other. These sub-documents are one of the great things about the document model. It allows us to apply the concept of "Data that's accessed together, is stored together" to our application.

We also have the ability to store information inside arrays which is another powerful feature of the document model.

These documents are JSON structured objects. Which is how most modern developers think of things. A person is an object that has various attributes, like a job title, address, etc.

This allows modern development practices to use the document model in a very intuitive way without having to break the data apart to put into tables and normalize things.

## MongoDB Atlas

You can host your MongoDB database locally but I've found that it is easier to host the database using MongoDB Atlas.

We will be using the free tier on MongoDB Atlas in this tutorial. The first step is to [make an account at the MongoDB Atlas website](#).

After you get logged in, click the green button to create a new project and then the green button to build a new cluster. The following screen shots show what the screens should look like.

The screenshot displays the MongoDB Atlas user interface. At the top, the 'mongoDB Atlas' logo and 'All Clusters' are visible. On the right, it shows 'Usage This Month: \$0.00' with a 'details' link and a user profile 'Beau'. The left sidebar contains a 'CONTEXT' section with a dropdown for 'MERN Tutorial' and an 'ORGANIZATION' section with links to 'Projects', 'Activity Feed', 'Access', 'Alerts', 'Billing', 'Settings', 'Docs', and 'Support'. The main area is titled 'Projects' and features a search bar with the placeholder 'Find a project...'. Below the search bar is a table with the following columns: 'Project Name', 'Clusters', 'Users', 'Teams', 'Alerts', and 'Actions'. A green 'New Project' button is located in the top right corner of the main area. At the bottom, the footer indicates 'System Status: All Good', 'Last Login: 73.18.141.130', and copyright information for MongoDB, Inc. 2019, along with links for Status, Terms, Privacy, Atlas Blog, and Contact Sales.

**mongoDB Atlas** All Clusters

Eastern Time (US & Canada) Usage This Month:\$0.00 [details](#) Beau

CONTEXT

test

MERN TUTORIAL > TEST

# Clusters

Overview Security

Find a cluster...

## Create a cluster

Choose your cloud provider, region, and specs.

Build a Cluster

Once your cluster is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#).

PROJECT

Clusters

Alerts

Backup

Access

Settings

Stitch

Charts

Docs

Support

System Status: **All Good** Last Login: 73.18.141.130

©2019 MongoDB, Inc. [Status](#) [Terms](#) [Privacy](#) [Atlas Blog](#) [Contact Sales](#)

The first step to configure the new cluster is to choose your Cloud Provider, the zone or region you want your data to be stored in.

Notice that some of the regions offer a Free Tier which is great for a sandbox environment.

## Create New Cluster

Welcome to MongoDB Atlas! We've recommended some of our most popular options, but feel free to customize your cluster to your needs. For more information, check our [documentation](#).

### Global Cluster Configuration















#### Cloud Provider & Region

GCP, Iowa (us-central1) ▾



Create a **free tier cluster** by selecting a region with **FREE TIER AVAILABLE** and choosing the **M0** cluster tier below.

★ recommended region ⓘ

NORTH AMERICA / SOUTH AMERICA	EUROPE / MIDDLE EAST / AFRICA	AUSTRALIA
<div> South Carolina (us-east1) ★</div>	<div> Belgium (europe-west1) ★ <div>FREE TIER AVAILABLE</div></div>	<div> Sydney (australia-southeast1) ★</div>
<div> N. Virginia (us-east4) ★</div>	<div> Finland (europe-north1) ★</div>	
<div> Los Angeles (us-west2) ★</div>	<div> London (europe-west2) ★</div>	<div><div>ASIA PACIFIC</div><div> Taiwan (asia-east1) ★ <div>FREE TIER AVAILABLE</div></div></div>
<div> Iowa (us-central1) ★ <div>FREE TIER AVAILABLE</div></div>	<div> Frankfurt (europe-west3) ★</div>	<div> Tokyo (asia-northeast1) ★</div>
<div> Oregon (us-west1) ★</div>	<div> Netherlands (europe-west4) ★</div>	<div><div> Singapore (asia-southeast1) ★ <div>FREE TIER AVAILABLE</div></div></div>

**FREE**

Free forever! Your M0 cluster is ideal for experimenting in a limited sandbox. You can upgrade to a production cluster anytime.

Cancel

Create Cluster

After the cluster is created, you will have to configure your security. The two things we are required to setup from a security standpoint are IP Whitelist addresses and a database user. For the IP Whitelist, just add your current IP address.

# Connect to Cluster0

Setup connection security

Choose a connection method

Connect

You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. [Read more](#)

**You can't connect yet.** Set up your firewall access and user security permission below.

## 1 Whitelist your connection IP address

Add Your Current IP Address

Add a Different IP Address

## 2 Create a MongoDB User

This first user will have [atlasAdmin](#) permissions for all clusters in this project.

Keep your credentials handy, you'll need them for the next step.

Username

ex. dbUser

Password

ex. dbUserPassword

Autogenerate Secure Password

SHOW

Create MongoDB User

Once those steps have been completed, we can move on and get our connection information.

There are a few different ways that we provide information to connect to MongoDB Atlas.

1. Through the MongoDB Shell, which is a command line interface.
2. With an application connection string, which is what we'll use.
3. Through MongoDB Compass which is a GUI tool for interacting with data stored in MongoDB.

Click the "Connect Your Application" button. You will see information on getting a connection string and then some connection examples for different languages.



## Connect to Cluster0

✓ Setup connection security

Choose a connection method

Connect

### Choose a connection method [View documentation](#)

See methods to add data and diagnostics in the [Command Line Tools](#) shortcut from within your cluster.



#### Connect with the Mongo Shell

Mongo Shell with TLS/SSL support is required



#### Connect Your Application

Get a connection string and view driver connection examples



#### Connect with MongoDB Compass

Download Compass to explore, visualize, and manipulate your data



Go Back

Close

## Connect to Cluster0

✓ Setup connection security

✓ Choose a connection method

Connect

### 1 Choose your driver version

DRIVER

Node.js

VERSION

3.0 or later

### 2 Add your connection string into your application code

Connection String Only

Full Driver Example

```
mongodb+srv://mean123:<password>@cluster0-91icu.gcp.mongodb.net
```

Copy

Replace **<password>** with the password for the *mean123* user.

When entering your password, make sure that any special characters are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

Go Back

Close

Later in the code, we will need the connection string and the password for the user that was created earlier.

## Initial set-up

Time to code! In this tutorial we'll be building up the code a little bit at a time, but you can see the completed code here: <https://github.com/beaucarnes/mern-exercise-tracker-mongodb>

Verify you have Node.js installed on your system by typing the following on the command line:

```
$ node -v
```

Note: Throughout this tutorial, any line that starts with **\$** means you should type everything after the **\$** at the command prompt in a terminal window. Do not type the **\$**. So for the previous line, just type **node -v**.

This will show what version of Node.js you have installed. If you do not have Node.js installed, make sure to [install it](#) before moving on.

Next, we'll create the initial React project by using *create-react-app*. The *npx* command allows us to run create-react-app without installing it first. Run this command:



```
$ npx create-react-app mern-exercise-tracker
```

This creates a directory containing the default React project template with all dependencies installed. exercise

Change into the newly created folder:

```
$ cd mern-exercise-tracker
```

Start the development web server by running the following command:

```
$ npm start
```

This starts the development server for the front end of the app. But before we work more on the front end, we'll create the back end and connect it to MongoDB Atlas.

## Back end

---

Inside the root folder ("mern-exercise-tracker"), create a new folder and change into the folder by running the following commands in the terminal:

```
$ mkdir backend$ cd backend
```

We'll create a *package.json* file inside the folder by running:

```
$ npm init -y
```

Now we can install a few dependencies:

```
$ npm install express cors mongoose dotenv
```

So what are these packages?

We've already discussed *Express*. It is a fast and lightweight web framework for Node.js.

Cross-origin resource sharing (CORS) allows AJAX requests to skip the Same-origin policy and access resources from remote hosts. The *cors* package provides an Express middleware that can enable CORS with different options.

And we already discussed *mongoose*. It makes interacting with MongoDB through Node.js simpler.

*dotenv* loads environment variables from a *.env* file into *process.env*. This makes development simpler. Instead of setting environment variables on our development machine, they can be stored in a file. We'll create the *.env* file later.

We'll install one final package globally. Run:

```
$ npm install -g nodemon
```

*nodemon* makes development easier. It is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

Time to create the backend server! Make a file named *server.js* inside the backend directory. In the server file, we'll create an Express server, attach the cors and express.json middleware (since we will be sending and receiving json), and make the server listen on port 5000. Add the following code to create a basic Node.js / Express server:

Now we can the server by using *nodemon*:

```
$ nodemon server
```

You should be able to see the server running in the terminal.

It's finally time to connect to our database in MongoDB Atlas. At the top of `server.js`, after the line `const cors = require('cors');`, add the following line to require `mongoose`:

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

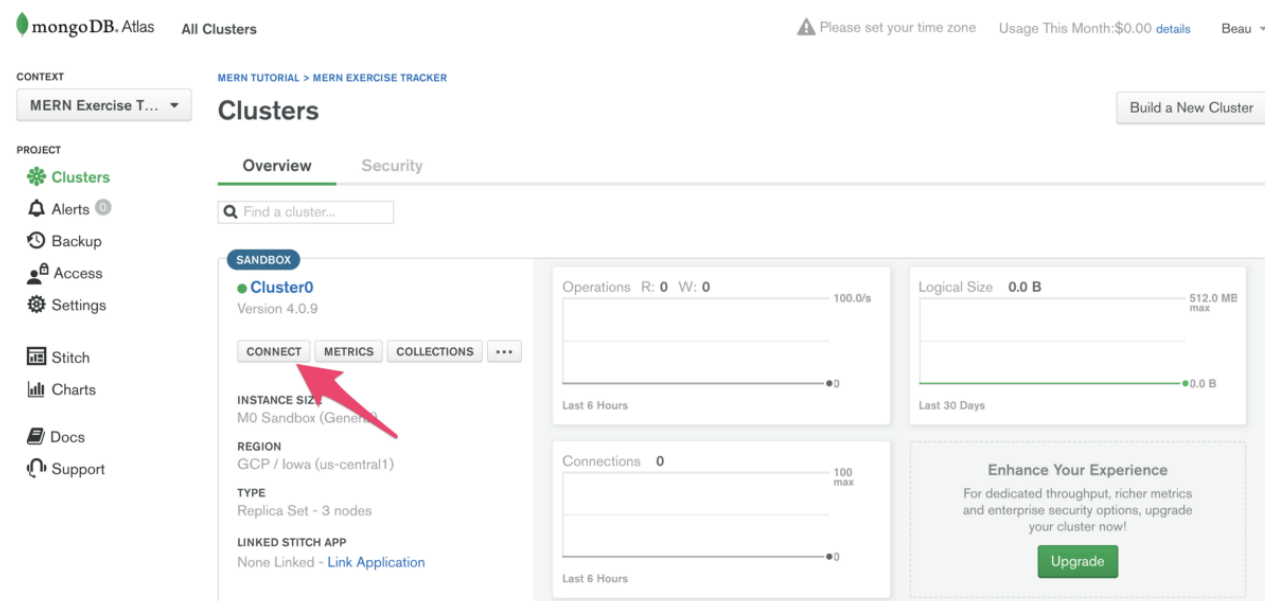
Now, after the line `app.use(express.json());`, add:

```
const uri = process.env.ATLAS_URI;mongoose.connect(uri, { useNewUrlParser: true, useCreateIndex: true });const connection = mongoose.connection;connection.once('open', () => { console.log("MongoDB database connection established successfully");})
```

The section `useNewUrlParser: true` is added because the MongoDB Node.js driver rewrote the tool it uses to parse MongoDB connection strings. Because this is such a big change, they put the new connection string parser behind a flag. The section `useCreateIndex: true` is similar. It is to deal with MongoDB deprecating the `ensureIndex()` function.

For the connection to work, we'll need to add the correct `ATLAS_URI` environment variable.

In the server directory, create a file named `.env`. Now, we need to get the uri. This is the connection string from MongoDB Atlas that was mentioned earlier. Here is how to get back to it from the MongoDB Atlas dashboard:





## Connect to Cluster0

✓ Setup connection security

Choose a connection method

Connect

**Choose a connection method** [View documentation](#)

See methods to add data and diagnostics in the [Command Line Tools](#) shortcut from within your cluster.



### Connect with the Mongo Shell

Mongo Shell with TLS/SSL support is required



### Connect Your Application

Get a connection string and view driver connection examples



### Connect with MongoDB Compass

Download Compass to explore, visualize, and manipulate your data



Go Back

Close

## Connect to Cluster0

✓ Setup connection security
✓ Choose a connection method
Connect

### 1 Choose your driver version

DRIVER	VERSION
Node.js	3.0 or later

### 2 Add your connection string into your application code

#### Connection String Only

#### Full Driver Example

```
mongodb+srv://mean123:<password>@cluster0-91icu.gcp.mongodb.net
```

Copy

Replace **<password>** with the password for the *mean123* user.  
When entering your password, make sure that any special characters are [URL encoded](#).

Having trouble connecting? [View our troubleshooting documentation](#)

[Go Back](#)
[Close](#)

In the `.env` file, type “ATLAS\_URI=” and then paste in the uri / connection string you just copied. It should look something like this:

```
ATLAS_URI=mongodb+srv://mean123:<password>-91icu.gcp.mongodb.net/test?retryWrites=true
```

Replace `<password>` with the password you set up for your user.

In the terminal running your server you should now see the line “MongoDB database connection established successfully”. You may have to restart the server first.

## Database Schema

Next we’ll create our database schema using Mongoose. We’ll have two entities: Exercises and Users.

Inside the backend folder, create a new folder named “models”. Inside that folder create two files named *exercise.model.js* and *user.model.js*.

Add the following code for the user model:

The User Schema only contains a single field: username. We added some validations to the username field. It is required, it must be unique, and it must be at least 3 characters long. Also, white space is trimmed off the end.

Now add the following code to the exercise model:

In the Exercise Schema, there are four fields. Since we don't use as much validation, each fits on its own line.

## Server API Endpoints

We now need to add the API endpoint routes so the server can be used to perform CRUD operations.

Inside the backend folder, create a new folder named "routes". Inside that folder create two files named *exercises.js* and *users.js*.

We'll come back to those files. First we'll tell the server to use the files we just created. Toward the end of *server.js*, right before the line `app.listen(port, function() {`, add:

```
const exercisesRouter = require('./routes/exercises');const usersRouter =  
require('./routes/users');app.use('/exercises', exercisesRouter);app.use('/users', usersRouter);
```

The first two lines load the routers from other files. Then the routers are added as middleware.

The server URL is `https://localhost:5000`. Now if you add "/exercises" or "/users" on the end it will load the endpoints defined in the corresponding router files. So let's build out those router files.

In the "users.js" file you just created, add the following code:

The first endpoint handles incoming HTTP GET requests on the `/users/` URL path. We call `Users.find()` to get a list of all the users from the database. The find method returns a promise. The results are returned in JSON format with `res.json(users)`.

The second endpoint handles incoming HTTP POST requests on the `/users/add/` URL path. The new username is part of the request body. After getting the username, we create a new instance of *User*. Finally, the new user is saved to the database with the `save()` method and we return "User added!"

Add these same two endpoints to "exercises.js". You will notice that this time we break out all four fields from the submitted data.

## Testing the server API

We'll add some more API endpoints soon. But first let's test the server API. We'll be using a tool called Insomnia. Another popular tool for this purpose is Postman.

Once you get Insomnia (or Postman) installed, create a new POST request using JSON.



New Request

Name

Exercise Tracker

POST

JSON

\* Tip: paste Curl command into URL afterwards to import it

Create

Enter the url: `http://localhost:5000/users/add`

Enter the JSON (feel free to use your own name):

```
{ "username": "beau" }
```

Finally, click “Send”. You should see the response “User added!”

Insomnia interface showing a POST request to `http://localhost:5000/users/add`. The request body is JSON: `{ "username": "beau" }`. The response is `200 OK` with status `TIME 235 ms` and `SIZE 13 B`. The response body is `"User added!"`. Red arrows point to the URL, JSON data, and the response body. Text overlays indicate: "1. Enter URL", "2. Enter JSON data", "3. Click 'Send' button", and "Successful response!".

We can send a GET request to “`https://localhost:5000/users/`” and get a list of users back. It should return the user we just added, with some additional metadata.

Insomnia interface showing a GET request to `http://localhost:5000/users/`. The response is `200 OK` with status `TIME 136 ms` and `SIZE 140 B`. The response body is a JSON array: `[ { "_id": "5cc76a5305a94a762ba41749", "username": "beau", "createdAt": "2019-04-29T21:19:15.187Z", "updatedAt": "2019-04-29T21:19:15.187Z", "__v": 0 } ]`. Red arrows point to the URL, the 'Body' tab, and the response body. A hand icon is visible in the background.

We can also see the user we just added on the MongoDB Atlas dashboard.

CONTEXT

MERN Exercise T... ▾

MERN TUTORIAL > MERN EXERCISE TRACKER

## Clusters

Build a New Cluster

PROJECT

- Clusters
- Alerts 0
- Backup
- Access
- Settings
- Stitch
- Charts
- Docs
- Support

Overview

Security

Find a cluster...

**SANDBOX**

● **Cluster0**  
Version 4.0.9

CONNECT METRICS **COLLECTIONS** ...

**INSTANCE SIZE**  
M0 Sandbox (General)

**REGION**  
GCP / Iowa (us-central1)

**TYPE**  
Replica Set - 3 nodes

**LINKED STITCH APP**  
None Linked - [Link Application](#)

Operations R: 0 W: 0 0.006/s

Last 6 Hours

Logical Size **32.1 KB** 512.0 MB max

Last 30 Days

Connections **1** 100 max

Last 6 Hours

**Enhance Your Experience**

For dedicated throughput, richer metrics and enterprise security options, upgrade your cluster now!

Upgrade

System Status: All Good Last Login: 73.18.141.130

©2019 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

mongoDB Atlas All Clusters

Eastern Time (US & Canada) ▾ Usage This Month:\$0.00 details Beau ▾

CONTEXT

MERN Exercise T... ▾

MERN TUTORIAL > MERN EXERCISE TRACKER > CLUSTERS

## Cluster0

VERSION 4.0.9 REGION Iowa (us-central1)

PROJECT

- Clusters
- Alerts 0
- Backup
- Access
- Settings
- Stitch
- Charts
- Docs
- Support

Overview

Real Time

Metrics

**Collections**

Command Line Tools

DATABASES: 1 COLLECTIONS: 1

+ Create Database

Q NAMESPACES

test

users

**test.users**

COLLECTION SIZE: 176B TOTAL DOCUMENTS: 2 INDEXES TOTAL SIZE: 64KB

Find Indexes

INSERT DOCUMENT

FILTER {"filter": "example"}

Find Reset

QUERY RESULTS 1-1 OF 1

```

_id: ObjectId("5cc76a5305a94a762ba41749")
username: "beau"
createdAt: 2019-04-29T21:19:15.187+00:00
updatedAt: 2019-04-29T21:19:15.187+00:00
_v: 0
                    
```

System Status: All Good Last Login: 73.18.141.130

©2019 MongoDB, Inc. Status Terms Privacy Atlas Blog Contact Sales

Let's add a few exercises. Use Insomnia to POST the following data to "<http://localhost:5000/exercises/add>" (update the username to the one you used).

```
{ "username": "beau", "description": "run", "duration": 5, "date": "2019-04-29T21:19:15.187Z"}
```

Then POST another exercise:

15/26

```
{ "username": "beau", "description": "bike ride", "duration": 20, "date": "2019-04-30T21:19:15.187Z"}
```

You should be able to see the data you just entered by sending a GET request to [“http://localhost:5000/exercises/”](http://localhost:5000/exercises/). Or check the MongoDB Atlas dashboard.

The screenshot shows the MongoDB Atlas interface for a cluster named 'Cluster0'. The left sidebar contains navigation options like Clusters, Alerts, Backup, Access, Settings, Stitch, Charts, Docs, and Support. The main panel displays the 'test.exercises' collection, showing two documents in a JSON format. The first document has a duration of 5 and a description of 'run'. The second document has a duration of 20 and a description of 'bike ride'. The interface includes tabs for Overview, Real Time, Metrics, Collections, and Command Line Tools. A 'REFRESH' button is visible in the top right of the collection view.

Now that we’ve tested everything, we will finish up the exercise routes. Add this code after the routes you already added in “exercises.js”.

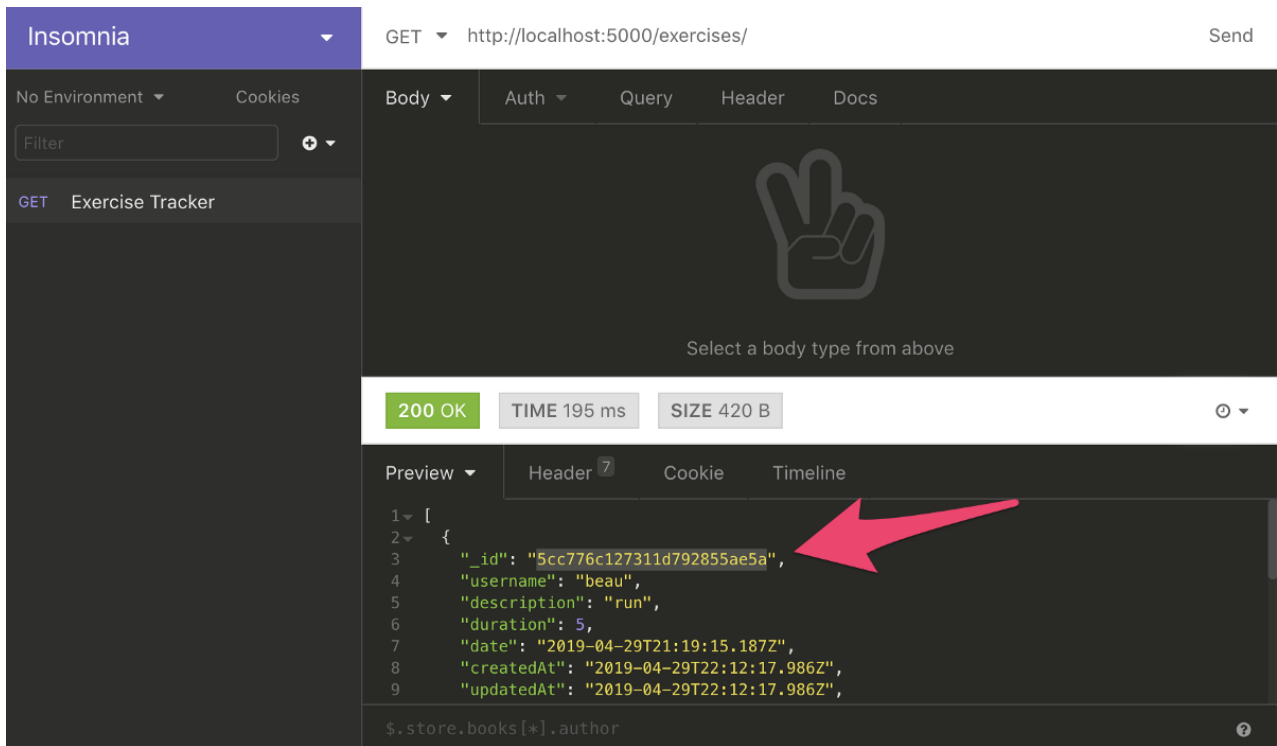
```
router.route('/:id').get((req, res) => { Exercise.findById(req.params.id) .then(exercise =>
res.json(exercise)) .catch(err => res.status(400).json('Error: ' +
err));});router.route('/:id').delete((req, res) => { Exercise.findByIdAndDelete(req.params.id)
.then(() => res.json('Exercise deleted.)) .catch(err => res.status(400).json('Error: ' +
err));});router.route('/update/:id').post((req, res) => { Exercise.findById(req.params.id)
.then(exercise => { exercise.username = req.body.username; exercise.description =
req.body.description; exercise.duration = Number(req.body.duration); exercise.date =
.parse(req.body.date); exercise.save() .then(() => res.json('Exercise updated!'))
.catch(err => res.status(400).json('Error: ' + err)); }) .catch(err =>
res.status(400).json('Error: ' + err));});
```

The `/:id` GET endpoint returns an exercise item given an *id*. The `/:id` DELETE endpoint deletes an exercise item given an *id*.

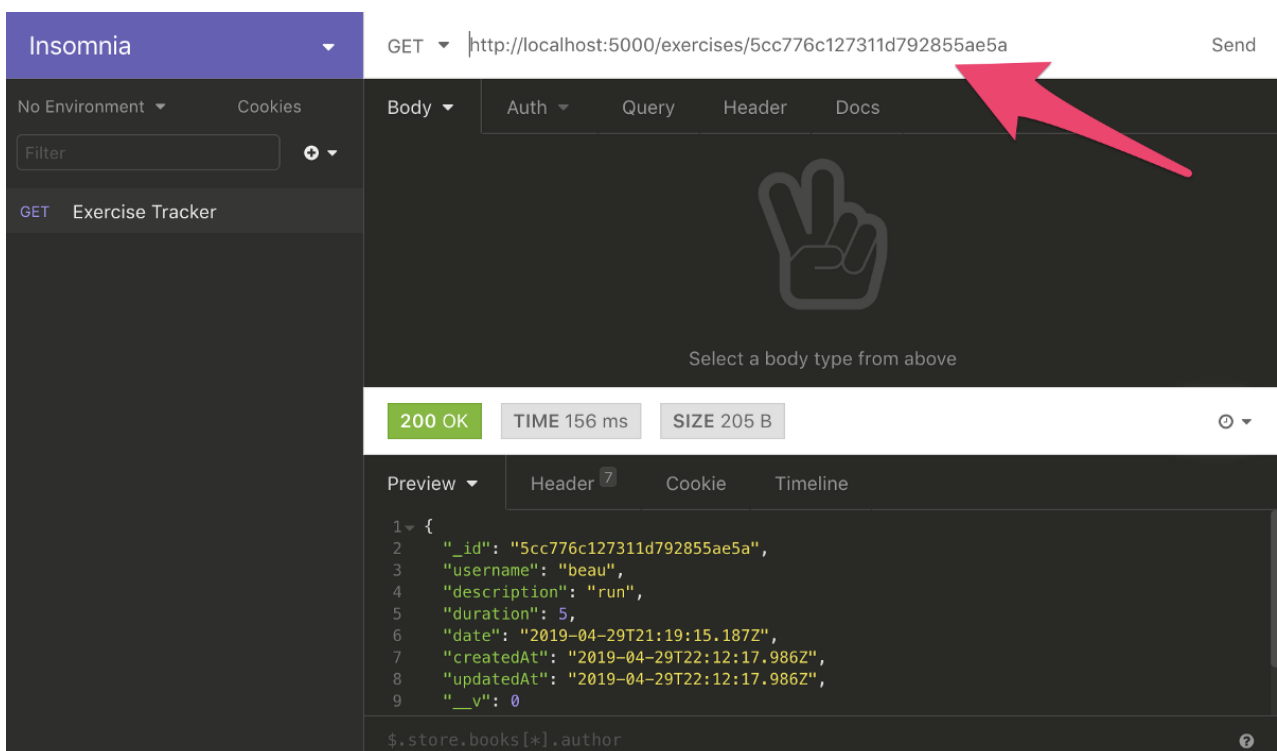
Finally, the `/update/:id` POST endpoint updates an existing exercise item. For this endpoint, we first retrieve the old exercise item from the database based on the *id*. Then, we set the exercise property values to what’s available in the request body. Finally, we call `exercise.save` to save the updated object in the database.

We can now test these endpoints with Insomnia. To test the first endpoint we just added, we need an *id*. Get the first id by sending a GET request to <https://localhost:5000/exercises/> . Copy the first *id*.

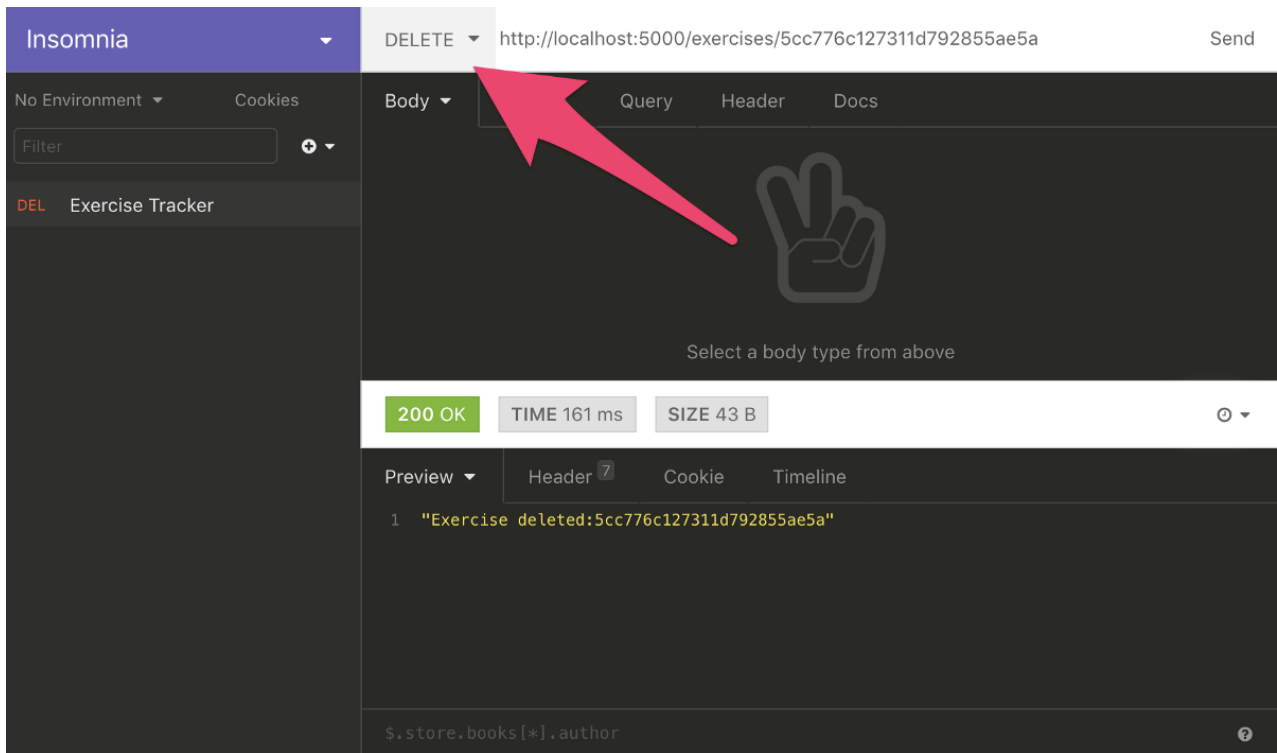




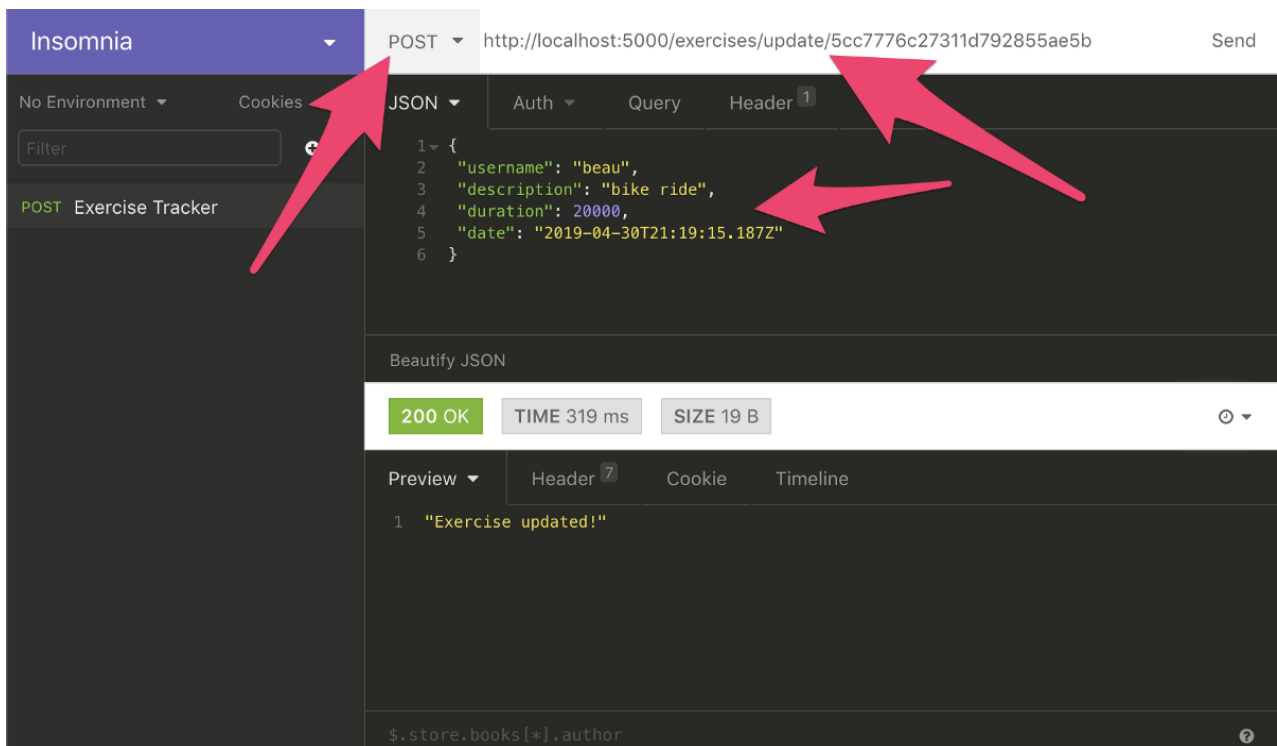
To test the `/:id` GET endpoint, paste that `id` to the end of the URL and send another GET request. You will see the exercise returned.



To delete the exercise, change GET to DELETE and send the request again.



Now we'll edit the other exercise. First, get the *id* the same way as before. Send a POST request to `https://localhost:5000/exercises/update/[YOUR ID]` (but change [YOUR ID] to the actual *id*.) We will just update the duration to 20000, but you will have to send all the fields at JSON. You can get the old data from the GET request.



## Frontend

The backend is complete and now it is time to start the frontend!

As was mentioned earlier, we'll use React for the frontend. I'll give a very brief overview of React components and JSX. To learn more about React, watch [this free 5 hour React course](#).

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”. Our project isn’t too complex but it will still give you a good introduction to React.

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Cereal</li>
          <li>Milk</li>
          <li>Bananas</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Beau" />
```

Example React component

We use components to tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components.

In the code image above, `ShoppingList` is a React component class. A component takes in parameters, called props (short for “properties”), and returns a hierarchy of views to display through the render method.

The render method returns a description of what you want to see on the screen. React takes the description and displays the result. In particular, render returns a React element, which is a lightweight description of what to render.

Most React developers use a special syntax called “JSX” which makes these structures easier to write. The `<div />` syntax is transformed at build time to `React.createElement('div')`.

JSX comes with the full power of JavaScript. You can put any JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.

Now let’s get back to our project.

Make sure the frontend development web server is running. If it’s not, you can start it with the following command in a terminal:

```
$ npm start
```

Next, we need to add the Bootstrap CSS framework to our project to make styling easier. Inside the project folder run the following command in a terminal:

```
$ npm install bootstrap
```

Now need to make sure that Bootstrap's CSS file is imported in App.js by adding the following line of code after the "import React" line:

```
import "bootstrap/dist/css/bootstrap.min.css";
```

We can get rid of most of the default code in App.js. All we need is:

Now let's set up React Router. Run the following command in a terminal:

```
$ npm install react-router-dom
```

Import React Router in App.js by adding this as the second line in the file:

```
import { BrowserRouter as Router, Route } from "react-router-dom";
```

Next, let's embed the JSX code in a `<Router></Router>` element. App.js should look like this:

Inside the `<Router>` element we will add the router configuration. Replace "Hello World" with:

There is a `<Route>` element for each route of the application. The *path* attribute sets the url path. The component is the code that will be loaded when a user goes to that path.

We now need to create five components: one for the Navbar and four for the routes. We'll actually create them in separate files.

But first let's import the files we will create into App.js right after the import for bootstrap. Here is the full code of App.js after the imports are added:

In the same directory as App.js, create a new directory called 'components'. Inside that directory create a file called 'navbar.component.js'. Here is the code for that file:

This is just the navbar from the Bootstrap documentation converted to work for our purposes.

Next, create four more files in the components directory with the following names:

- exercises-list.component.js
- edit-exercise.component.js
- create-exercise.component.js
- create-user.component.js

At this point we will create stubs for these components so we can see if everything works so far.

*exercises-list.component.js:*

*edit-exercise.component.js:*

*create-exercise.component.js:*

*create-user.component.js:*

Now you should be able to test the project in the browser. When you click the buttons in the navigation bar, you should see different text appear.

## The Create Exercise Component

---

We'll create the Create Exercise Component now. This will allow us to add exercises into the database. Add a constructor to 'create-exercise.component.js'.

In JavaScript classes, you need to always call `super` when defining the constructor of a subclass. All React component classes that have a constructor should start it with a `super(props)` call.

We'll set the initial state of the component by assigning an object to `this.state`. The properties of state will correspond to the fields in the MongoDB document. Here is how to setup the constructor:

```
constructor(props) { super(props); this.state = { username: '', description: '', duration: 0, date: new Date(), users: [] } }
```

Now we need to add methods which can be used to update the state properties:

```
onChangeUsername(e) { this.setState({ username: e.target.value }); } onChangeDescription(e) { this.setState({ description: e.target.value }); } onChangeDuration(e) { this.setState({ duration: e.target.value }); } onChangeDate(date) { this.setState({ date: date }); }
```

You will notice the `onChangeDate` method looks a little different than the others. This is because of the date picker library we will be adding later.

After those methods, we'll add one to handle the submit event of the form which we will add later:

```
onSubmit(e) { e.preventDefault(); const exercise = { username: this.state.username, description: this.state.description, duration: this.state.duration, date: this.state.date, }; console.log(exercise); window.location = '/'; }
```

The `e.preventDefault()` prevents the default HTML form submit behavior from taking place.

Right now, we're just logging the exercise to the console. But once we create the backend, we'll update this to interact with the backend server.

After the form is submitted, the location is updated so the user is taken back to the home page.

Now we'll make sure `this` works properly in our methods, we need to bind the methods to `this`. Add these lines to the constructor:

```
this.onChangeUsername = this.onChangeUsername.bind(this); this.onChangeDescription = this.onChangeDescription.bind(this); this.onChangeDuration = this.onChangeDuration.bind(this); this.onChangeDate = this.onChangeDate.bind(this); this.onSubmit = this.onSubmit.bind(this);
```

We are just about to add the form that will allow people to enter exercises. But first we must add one final method. People filling out the form must select the user associated with the exercise from a drop down list.

Eventually, the user list will come directly from the MongoDB database. For now, though, we'll hardcode a single user. Add the following method, right after the constructor:

```
componentDidMount() { this.setState({ users: ['test user'], username: 'test user' }); }
```

The `componentDidMount()` method is part of the React life cycle and is invoked immediately after a component is mounted.

The final thing we'll do in this file for now is add the form code. Here is the complete file so far, including the code for the form:

Notice there is an `onChange` event for all the form elements that calls the corresponding methods.

Also notice that the final form element has a `DatePicker` component. At the top of the file we import the `DatePicker` component and the associated CSS. For the `DatePicker` component to work, we need to install the package via NPM. Run this in the terminal:

```
$ npm install react-datepicker
```

You can now test out the app so far. If you add an exercise, you will see it logged to the console.

Now that we've created that component, we'll create *create-user.component.js*. That one is much simpler.

First, after the line `export default class CreateUser extends Component {` add the constructor with *this* bindings:

```
constructor(props) { super(props); this.onChangeUsername = this.onChangeUsername.bind(this);
this.onSubmit = this.onSubmit.bind(this); this.state = { username: '' };
```

Now add the methods to change the username and submit the form:

```
onChangeUsername(e) { this.setState({ username: e.target.value });}onSubmit(e) {
e.preventDefault(); const newUser = { username: this.state.username, };

  console.log(newUser);

  this.setState({ username: '' })}
```

Finally, at the end in the return statement, delete the current code and add:

```
<div> <h3>Create New User</h3> <form onSubmit={this.onSubmit}> <div className="form-group">
<label>Username: </label> <input type="text" required className="form-control"
value={this.state.username} onChange={this.onChangeUsername} /> </div> <div
className="form-group"> <input type="submit" value="Create User" className="btn btn-primary" />
</div> </form></div>
```

You can now submit new users and the name will show up in the console.

## Connecting Front to Back

---

So far, the frontend and backend of our app are separate. We are now going to connect the two by causing our frontend to send HTTP request to the server endpoints on the backend.

We'll use the Axios library to send HTTP requests to our backend. Install it with the following command in your terminal:

```
$ npm install axios
```

Now we'll complete the implementation of the CreateUser component and send data to the backend.

In *create-user.component.ts*, after the "import React..." line, add:

```
import axios from 'axios';
```

To connect our code to the backend, we just need to add a single line to the *onSubmit* method. After `console.log(newUser);` add:

```
axios.post('/', newUser) .then(res => console.log(res.data));
```

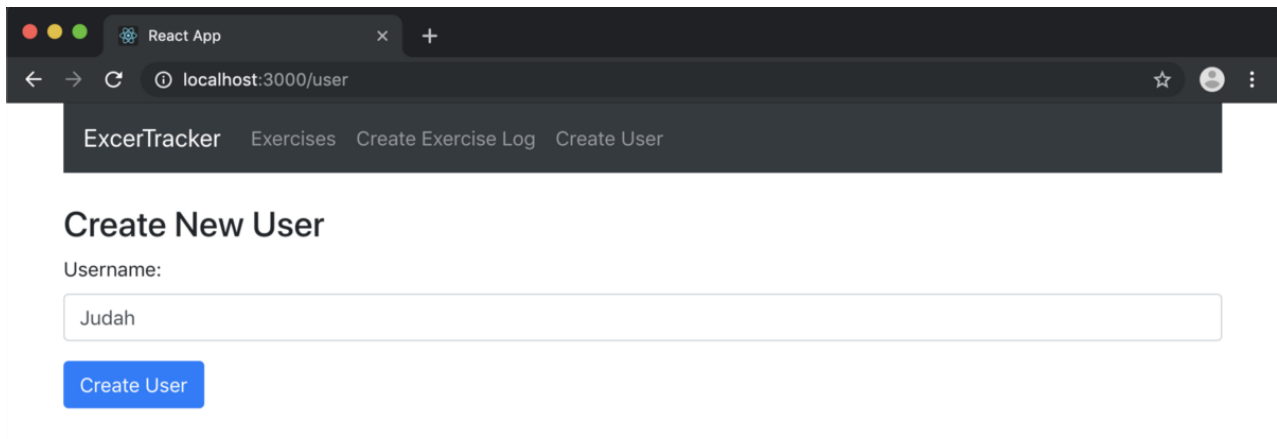
The *axios.post* method sends an HTTP POST request to the backend endpoint `http://localhost:5000/users/add`. This endpoint is expecting a JSON object in the request body so we passed in the *newUser* object as a second argument.

Now we can test things out.

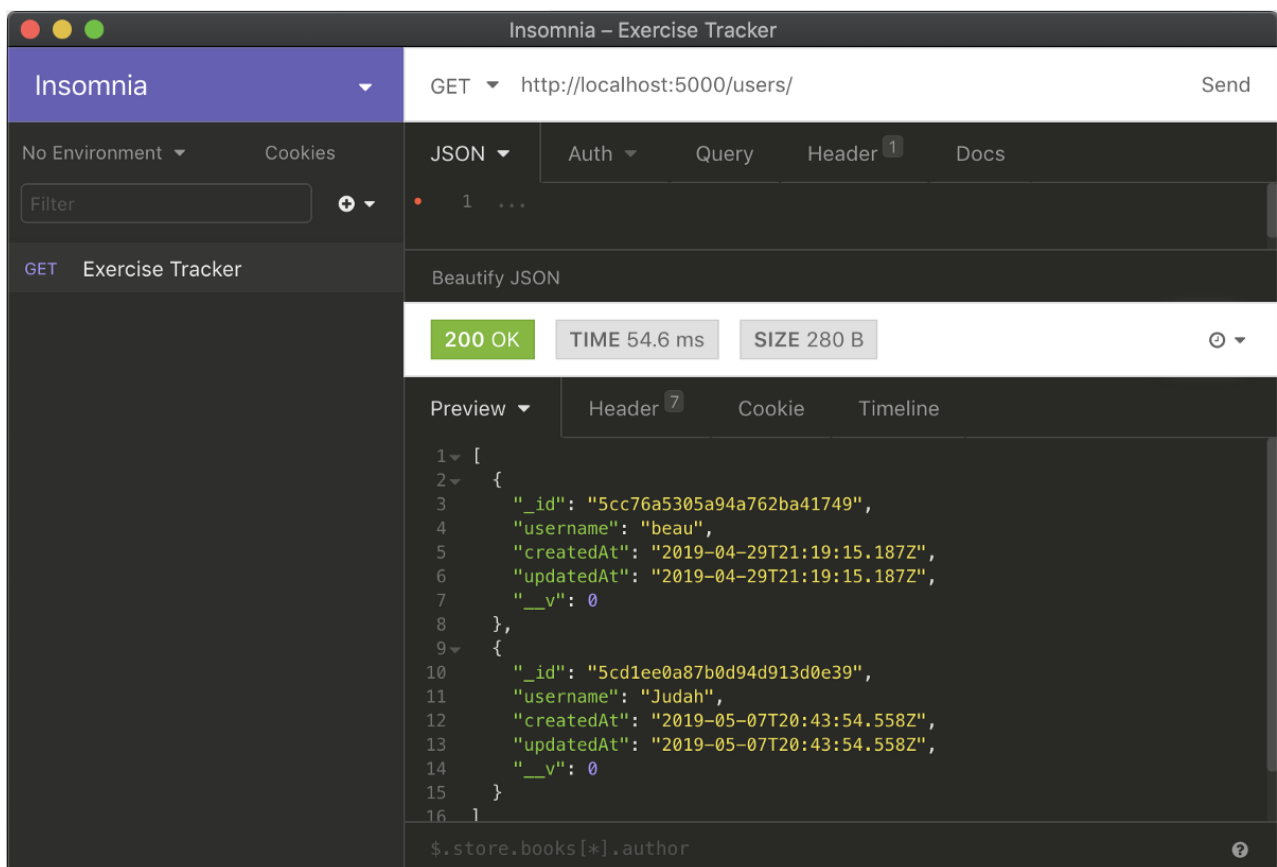
If your backend server is not still running, start it with the following command in the `backend` directory.

```
$ nodemon server
```

Point your web browser to `localhost:3000/user` and try adding a new username.



You can now see the new user using Insomnia or the MongoDB Atlas dashboard.



The screenshot shows the MongoDB Atlas interface for 'Cluster0'. The left sidebar contains navigation links: Clusters, Alerts, Backup, Access, Settings, Stitch, Charts, Docs, and Support. The main content area is titled 'test.users' and shows collection details: COLLECTION SIZE: 177B, TOTAL DOCUMENTS: 2, INDEXES TOTAL SIZE: 72KB. A 'Find' button is present, and the query results display two documents:

```

{
  "_id": ObjectId("5cc76a5305a94a762ba41749"),
  "username": "beau",
  "createdAt": 2019-04-29T21:19:15.187+00:00,
  "updatedAt": 2019-04-29T21:19:15.187+00:00,
  "__v": 0
}
{
  "_id": ObjectId("5cd1ee0a87b0d94d913d0e39"),
  "username": "Judah",
  "createdAt": 2019-05-07T20:43:54.558+00:00,
  "updatedAt": 2019-05-07T20:43:54.558+00:00,
  "__v": 0
}

```

Now we'll complete the implementation of the CreateExercise component to send data to the backend.

In `create-exercise.component.ts`, after the "import React..." line, add:

```
import axios from 'axios';
```

To connect our code to the backend, we just need to add a single line to the `onSubmit` method. After `console.log(exercise);` add:

```
axios.post('/', exercise) .then(res => console.log(res.data));
```

In this file, we must also update the `componentWillMount()` method. We will get the list of users from the database to add to the users dropdown menu in the form. Delete the current contents and add the following code:

```

axios.get('/') .then(response => {
  if (response.data.length > 0) {
    this.setState({
      users: response.data.map(user => user.username),
      username: response.data[0].username
    });
  }
}).catch((error) => {
  console.log(error);
});

```

You will see that we use the data returned from the database to set the state of `users` and `username`. `username` is automatically set to the first user in the database.

You can now test things out by adding an exercise and checking Insomnia or the MongoDB dashboard to see if it shows up.



React App

localhost:3000/create

ExcerTracker Exercises Create Exercise Log Create User

## Create New Exercise Log

Username:  
beau

Description:  
walk

Duration (in minutes):  
99

Date:  
05/07/2019

Create Exercise Log

Now we'll complete the `ExercisesList` component. At the beginning of `exercises-list.component.js`, after the `import React...` line, add:

```
import { Link } from 'react-router-dom';import axios from 'axios';
```

After the line `export default class ExercisesList extends Component {`, add a constructor to initialize the state with an empty `exercises` array and to bind `this` to the method we are about to create.

```
constructor(props) { super(props); this.deleteExercise = this.deleteExercise.bind(this); this.state = {exercises: []};}
```

Get the list of exercises from the database by adding this after the constructor:

```
componentDidMount() { axios.get('/') .then(response => { this.setState({ exercises: response.data }); }) .catch((error) => { console.log(error); })}
```

The code will run before the page is rendered and add the list of exercises to the state. The `axios.get` method accesses the `/exercises` endpoint. Then we assign `response.data` to the `exercises` property of the component's state object with the `this.setState` method.

This component will allow users to delete exercises. Add the method for that feature:

```
deleteExercise(id) { axios.delete('/') .then(res => console.log(res.data)); this.setState({ exercises: this.state.exercises.filter(el => el._id !== id) })}
```

We use the `axios.delete` method, then we update the state of exercises and filter out the exercise that was deleted.

Now change the `return` statement of the `render` function with the following JSX code:

```
<div> <h3>Logged Exercises</h3> <table className="table"> <thead className="thead-light">
<tr> <th>Username</th> <th>Description</th> <th>Duration</th> <th>Date</th>
<th>Actions</th> </tr> </thead> <tbody> { this.exerciseList() } </tbody> </table>
</div>
```

The exercises will appear in a table on the page. You'll notice that the body of the table just calls the `exerciseList()` method. We need to implement that method to return the rows of the table. Directly above the render function add:

```
exerciseList() { return this.state.exercises.map(currentexercise => { return <Exercise exercise={currentexercise} deleteExercise={this.deleteExercise} key={currentexercise._id}/>; })}
```

This method iterates through the list of exercise items by using the map function. Each exercise item is output with the Exercise component. We'll implement the exercise component next. The current exercise item is assigned to the exercise property of this component.

At the top of the file, after the import statements, add:

```
const Exercise = props => ( <tr> <td>{props.exercise.username}</td> <td>{props.exercise.description}</td> <td>{props.exercise.duration}</td> <td>{props.exercise.date.substring(0,10)}</td> <td> <Link to= {"/edit/"+props.exercise._id}>edit</Link> | <a href="#" onClick={() => { props.deleteExercise(props.exercise._id) }}>delete</a> </td> </tr>)
```

The Exercise component is implemented as a functional React component. The key thing that makes this type of component different from a class component is the lack of state and lifecycle methods. If all you need to do is to accept props and return JSX, use a functional component instead of a class component.

It outputs a table row with the values of the properties of the exercise item passed into the component. You'll notice that in the 'actions' column we output two links. One link goes to the edit route and the other calls the deleteExercise method.

We just linked to the Edit route. The final thing we need to do is to implement the EditExercise component. It is very similar to the CreateExercise component with just a few minor changes.

Update all the code in `edit-exercise.component.js` to the following code. The main differences from the CreateExercise component are the `componentDidMount` and `onSubmit` methods.

In the `componentDidMount` method we use `axios.get` to get the current exercise from the database and load the data into the state variables. The only difference in the `onSubmit` method is that the data is posted to the update route.

## Conclusion

---

We're done! We now have a fully functional MERN exercise tracker app using MongoDB Atlas. Test it and consider doing some refactoring.

As a reminder, you can access the completed code here: <https://github.com/beaucarnes/mern-exercise-tracker-mongodb>

*Note: This article was sponsored by MongoDB.*