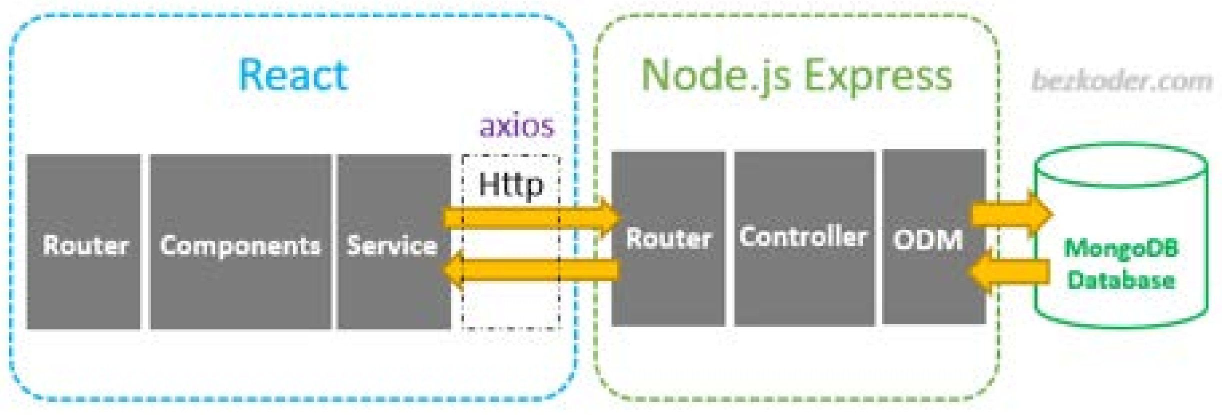
Session 12 – Notes – A MERN Application



A common practice for full stack stack is to start with creating a backend that can perform database operations. In this part, we are going to continue with setting up our backend and connecting it to MongoDB, but this time we create a cloud based database and create setup a CRUD opetation suited for a specific application. The steps are outlined below:

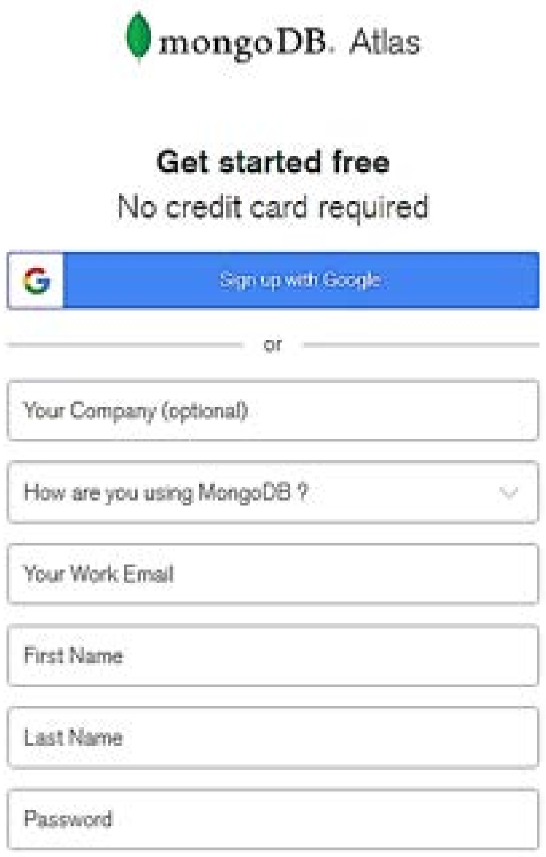
- Create a directory **tracker**
- **npx create-react-app tracker**
- Create a MongoDB account and login to https://www.mongodb.com/cloud/Atlas

Getting Started With MongoDB Atlas

mongoDB. Atlas

- The fastest, easiest way to get started with MongoDB.
- https://www.mongodb.com/cloud/atlas

For MongoDB Atlas, first create an account and sign in



mongoDB. Atlas

Get started free

No credit card required

Sign up with Google

or

Your Company (optional)

How are you using MongoDB ?

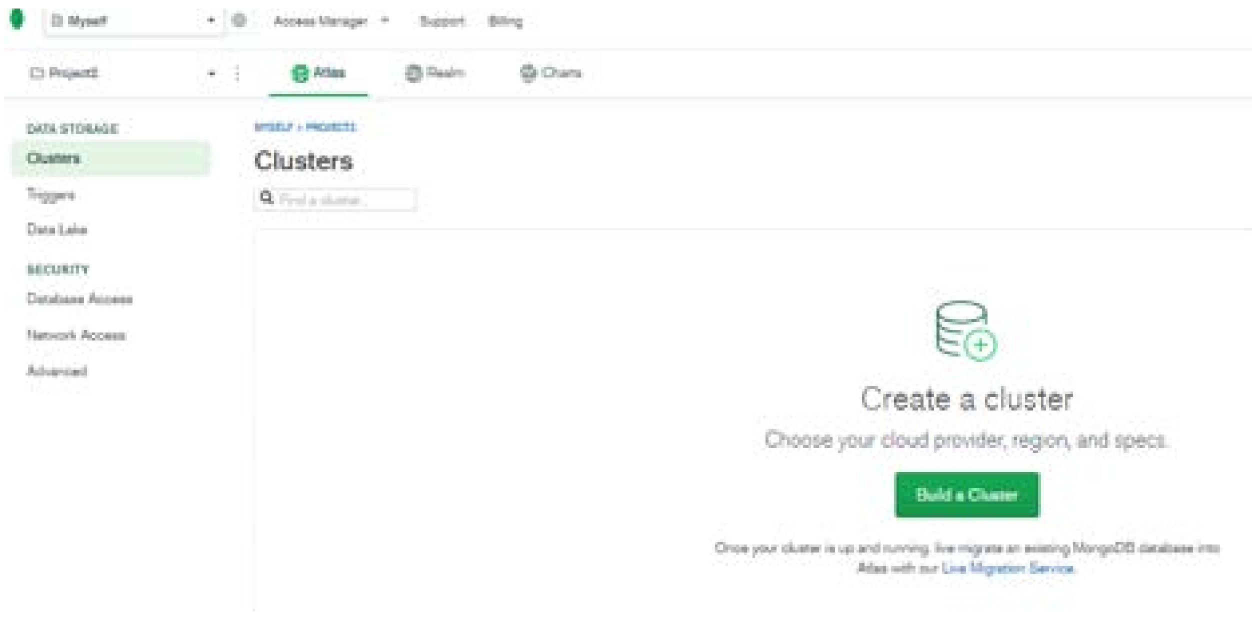
Your Work Email

First Name

Last Name

Password

First time you login, you need to create a cluster

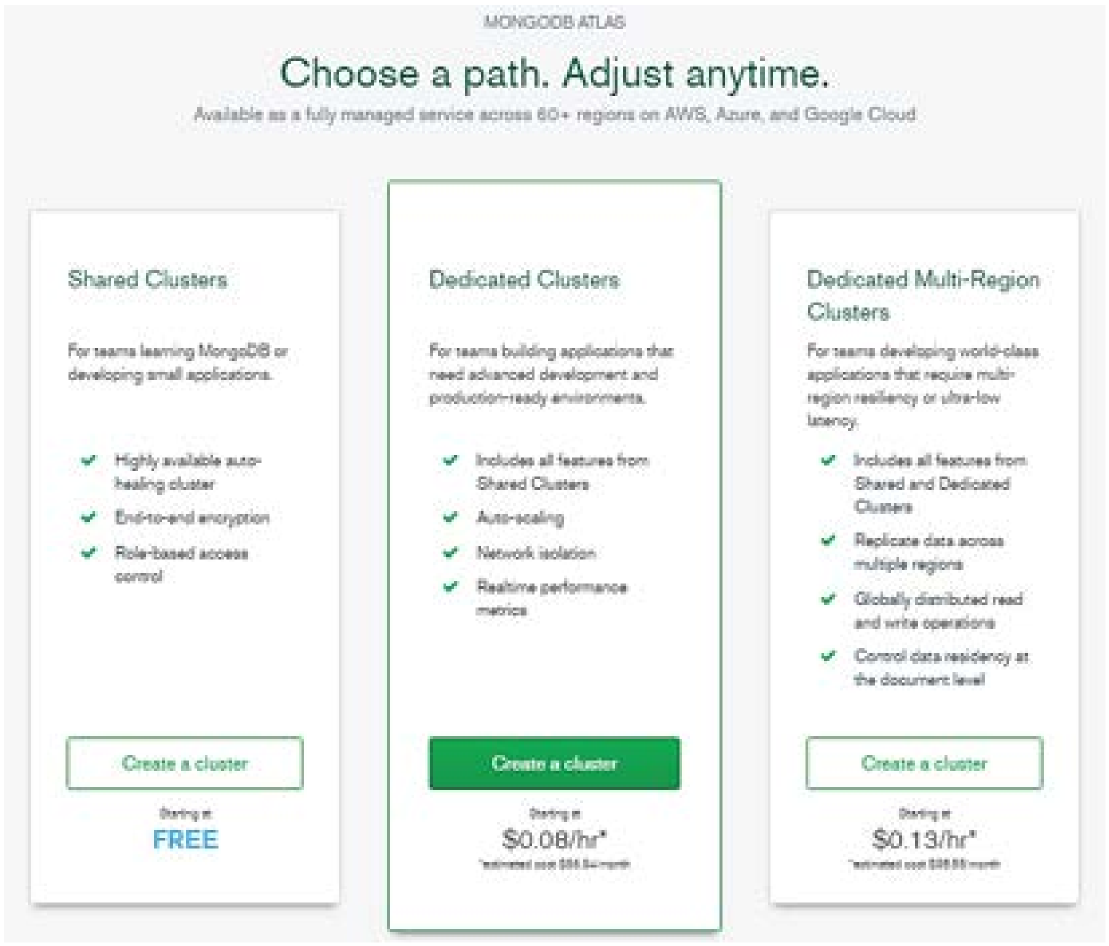


Clusters

Create a cluster

Choose your cloud provider, region, and specs.

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



MONGODB ATLAS

Choose a path. Adjust anytime.

Available as a fully managed service across 60+ regions on AWS, Azure, and Google Cloud

Shared Clusters

For teams learning MongoDB or developing small applications.

- Highly available auto-healing cluster
- End-to-end encryption
- Role-based access control

Create a cluster

Starting at
FREE

Dedicated Clusters

For teams building applications that need advanced development and production-ready environments.

- Includes all features from Shared Clusters
- Auto-scaling
- Network isolation
- Realtime performance metrics

Create a cluster

Starting at
$0.08/hr*
*estimated cost $56.94/month

Dedicated Multi-Region Clusters

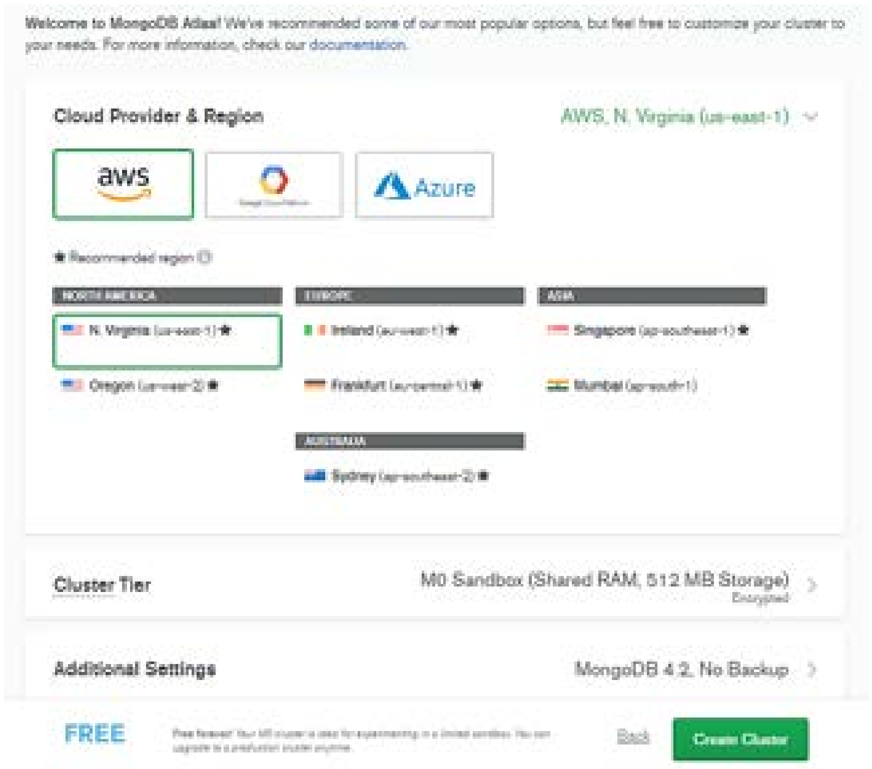
For teams developing world-class applications that require multi-region resiliency or ultra-low latency.

- Includes all features from Shared and Dedicated Clusters
- Replicate data across multiple regions
- Globally distributed read and write operations
- Control data residency at the document level

Create a cluster

Starting at
$0.13/hr*
*estimated cost $98.55/month

Create a 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.

Cloud Provider & Region — AWS, N. Virginia (us-east-1)

★ Recommended region

NORTH AMERICA	EUROPE	ASIA
N. Virginia (us-east-1) ★	Ireland (eu-west-1) ★	Singapore (ap-southeast-1) ★
Oregon (us-west-2) ★	Frankfurt (eu-central-1) ★	Mumbai (ap-south-1)

AUSTRALIA

Sydney (ap-southeast-2) ★

Cluster Tier — M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted

Additional Settings — MongoDB 4.2, No Backup

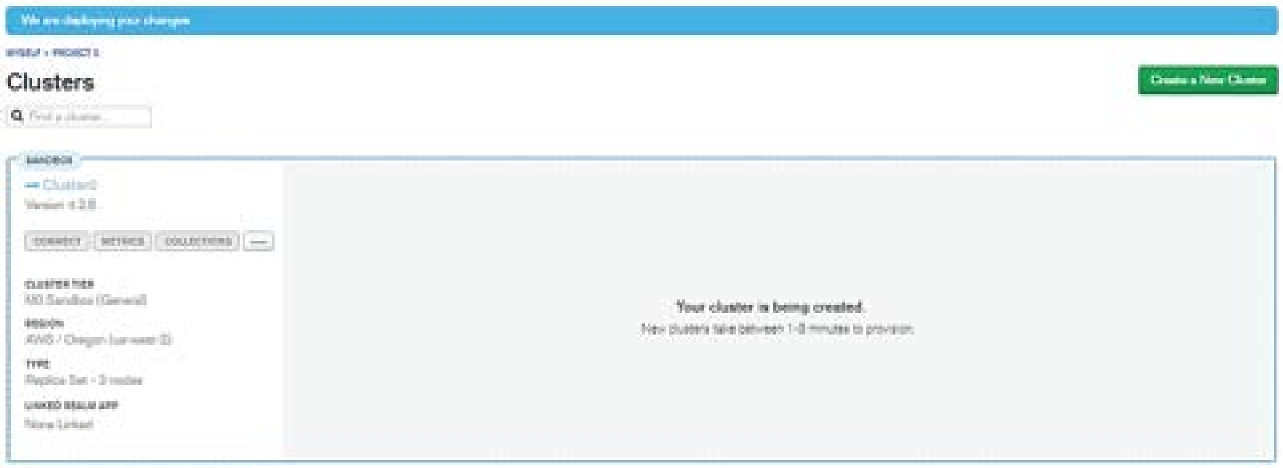
FREE [illegible]

It takes a few minutes to create the 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.



We are deploying your changes

Clusters

SANDBOX

Cluster0
Version 4.2.8

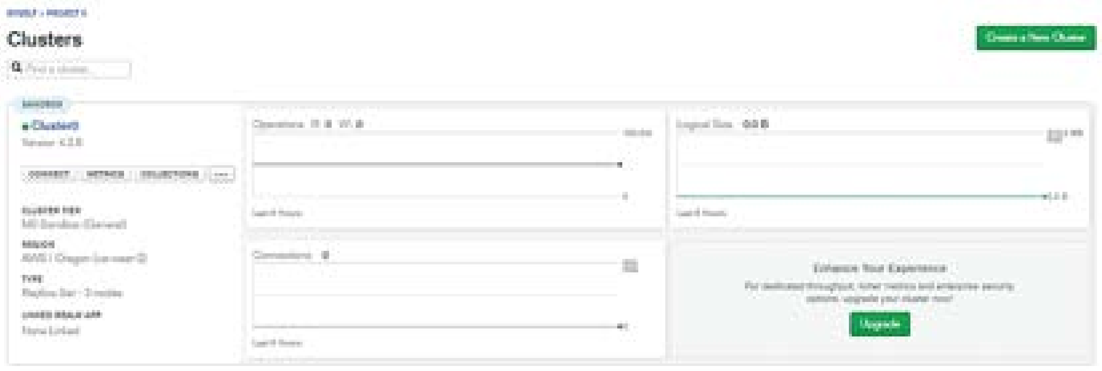
CLUSTER TIER
M0 Sandbox (General)

REGION
AWS / Oregon (us-west-2)

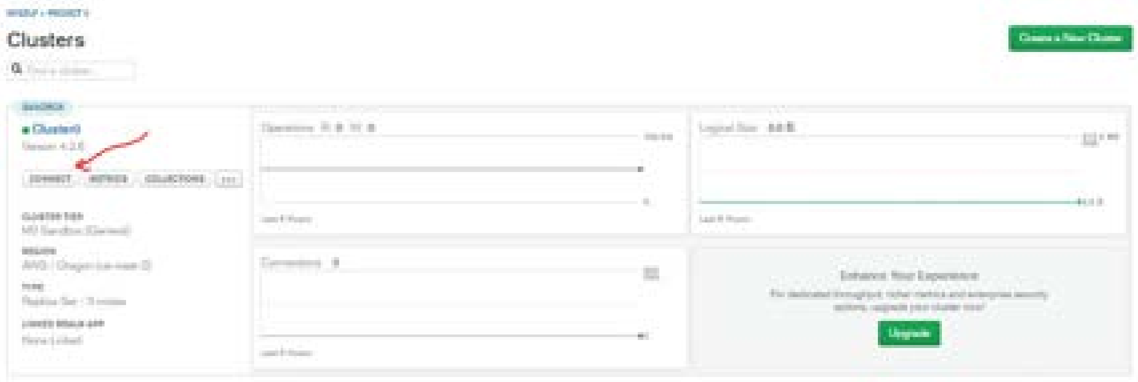
TYPE
Replica Set - 3 nodes

LINKED REALM APP
None Linked

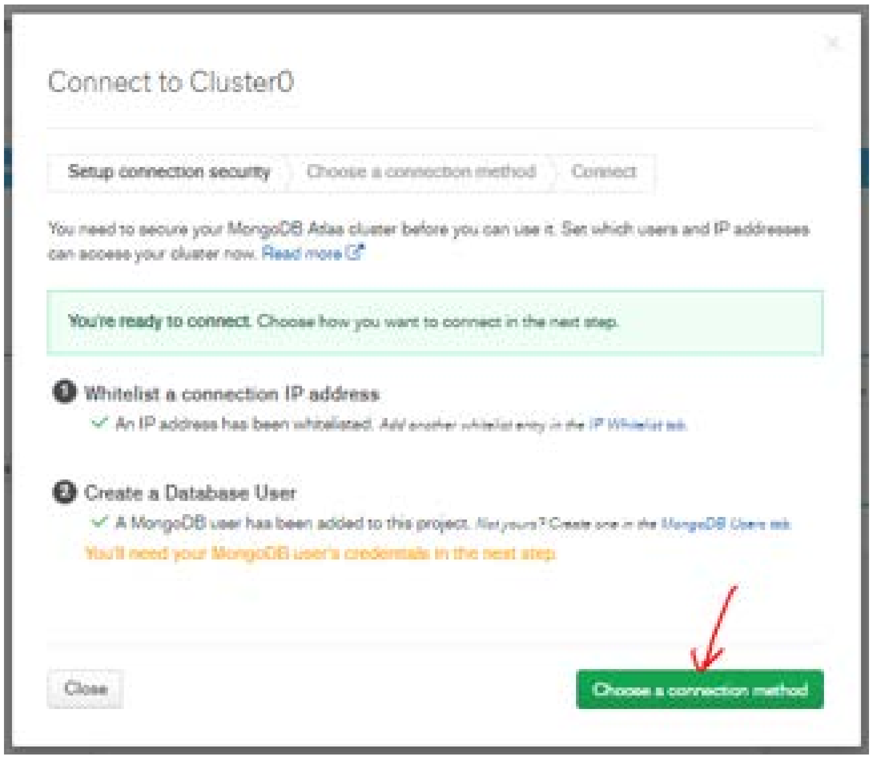
Your cluster is being created.
New clusters take between 1-3 minutes to provision.



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



- Now add a user name and a password of your choice (e.g., rezadb, 12345)



Connect to Cluster0

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're ready to connect. Choose how you want to connect in the next step.

1 Whitelist a connection IP address

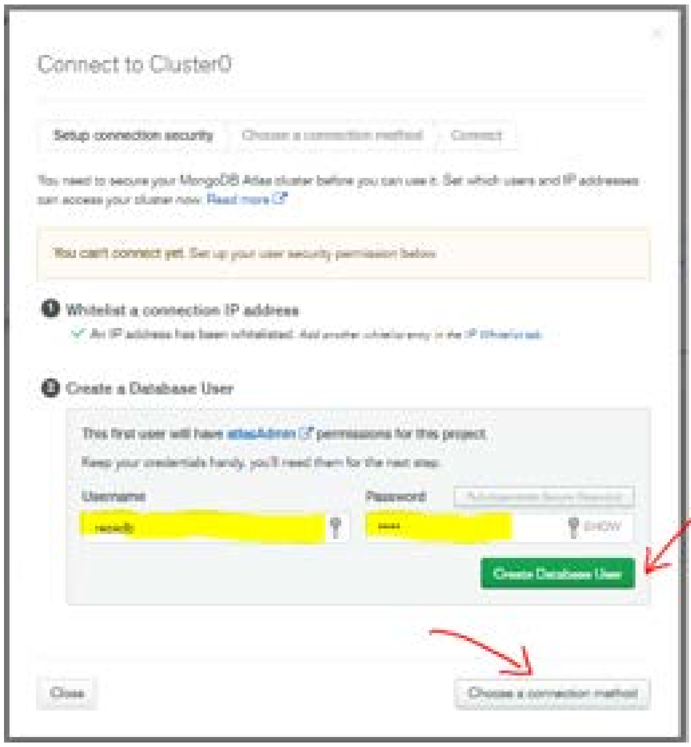
✓ An IP address has been whitelisted. *Add another whitelist entry in the IP Whitelist tab.*

2 Create a Database User

✓ A MongoDB user has been added to this project. *Not yours? Create one in the MongoDB Users tab.*

You'll need your MongoDB user's credentials in the next step.

-



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 user security permission below.

1 Whitelist a connection IP address

✓ An IP address has been whitelisted. Add another whitelist entry in the IP Whitelist tab.

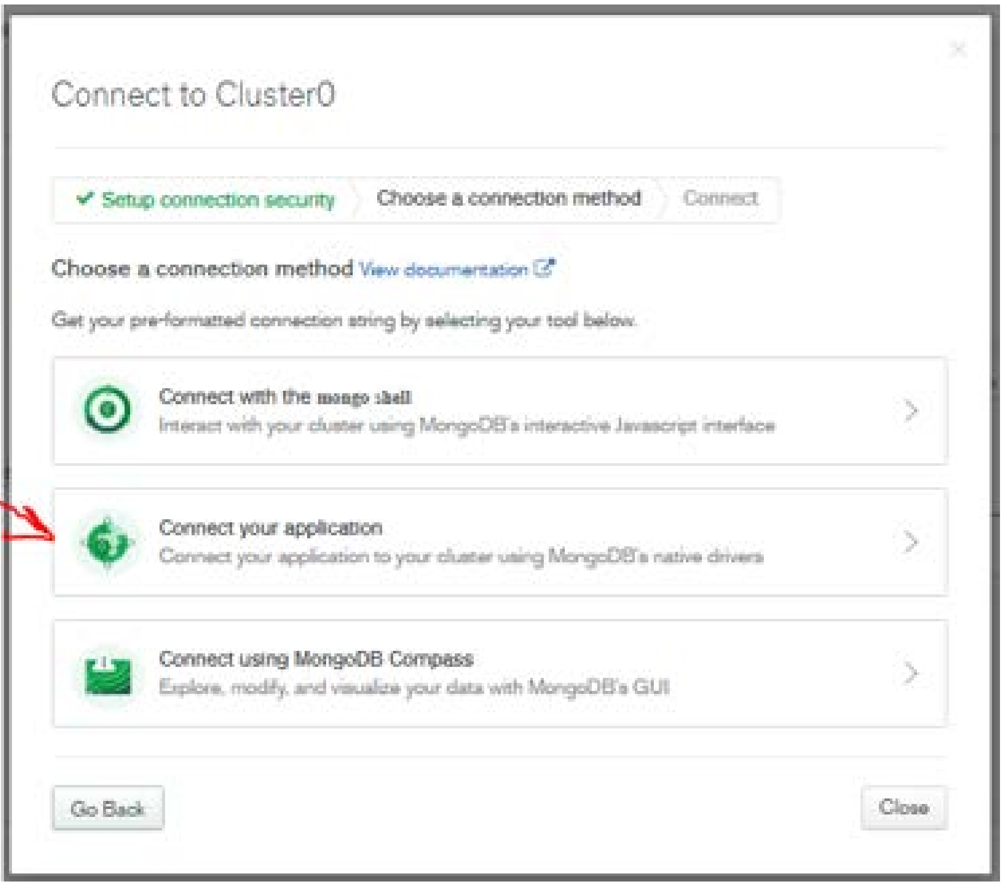
2 Create a Database User

This first user will have atlasAdmin permissions for this project.

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

Username: rezedb

Password: ••••



Connect to Cluster0

✓ Setup connection security | Choose a connection method | Connect

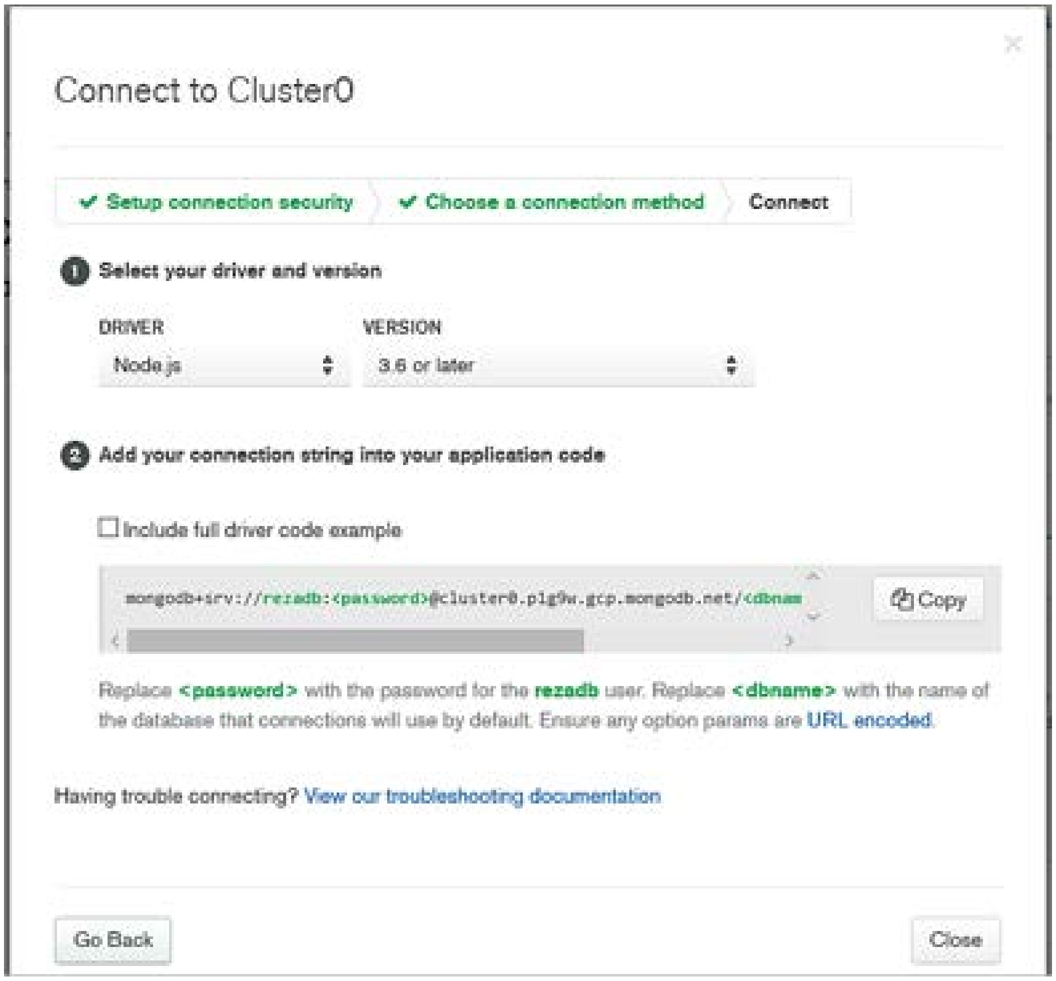
Choose a connection method View documentation

Get your pre-formatted connection string by selecting your tool below.

Connect with the mongo shell
Interact with your cluster using MongoDB's interactive Javascript interface

Connect your application
Connect your application to your cluster using MongoDB's native drivers

Connect using MongoDB Compass
Explore, modify, and visualize your data with MongoDB's GUI



Connect to Cluster0

1 Select your driver and version

DRIVER: Node.js

VERSION: 3.6 or later

2 Add your connection string into your application code

☐ Include full driver code example

```
mongodb+srv://rezadb:<password>@cluster0.p1g9w.gcp.mongodb.net/<dbnam
```

Replace <password> with the password for the rezadb user. Replace <dbname> with the name of the database that connections will use by default. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

Initial set-up

- npx create-react-app tracker
- cd tracker

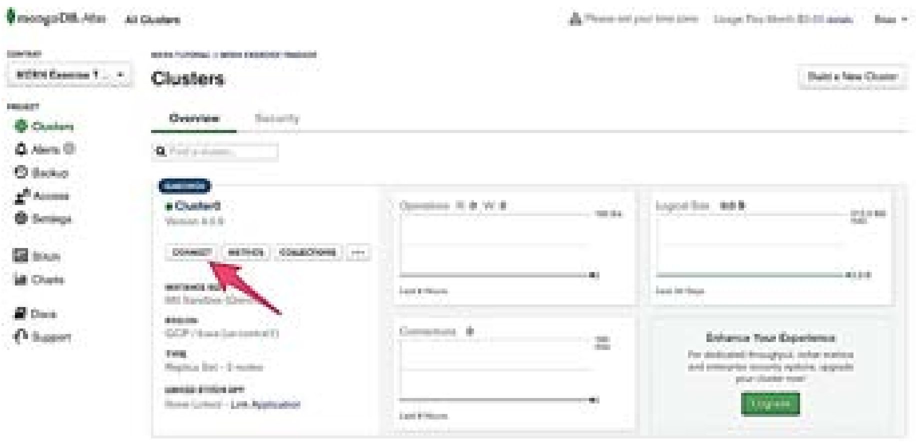
- first create the backend and connecting it to Mongodb google cloud … so **mkdir backend**
- **cd backend**
- **npm init -y**
- **npm install express cors mongoose dotenv**
- create **server.js** in backend directory and add the following code
- Cross-origin resource sharing (CORS) allows AJAX requests to skip the Same-origin policy and access resources from remote hosts. The cors package provides an Express middleware that can 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.

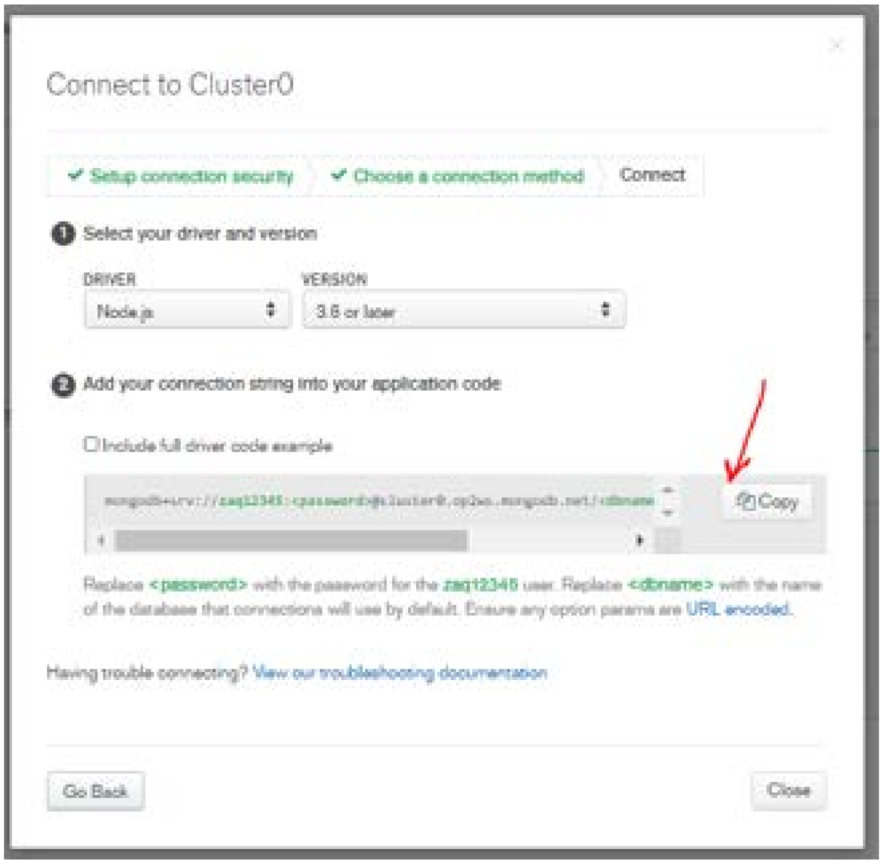
```
const express = require('express');
const cors = require('cors');
const mongoose = require('mongoose');
// require('dotenv').config();

const app = express();
const port = process.env.PORT || 5000;

app.use(cors());
app.use(express.json());

// app.use('/todos', todoRouter);

app.listen(port, () => {
    console.log(`Server is running on port: ${port}`);
});
```

- you can run the server while coding **nodemon server**
- add the following code after app.use(express.json());

```
app.use(cors());
app.use(express.json());

const uri = process.env.ATLAS_URI;
// const uri = "mongodb+srv://rezadb:zaq54321@cluster0.op2ws.mongodb.net/test?retryWrites=
true&w=majority";
mongoose.connect(uri, { useNewUrlParser: true,  useUnifiedTopology: true, useCreateIndex: true});
const connection = mongoose.connection;
connection.once("open", () => {
  console.log("MongoDB database connection established successfully");
});
```

- uri is coming from MongoDB cluster connection …. Copy and use it… or copy the whole code





Connect to Cluster0

✔ Setup connection security ✔ Choose a connection method Connect

1 Select your driver and version

DRIVER: Node.js

VERSION: 3.6 or later

2 Add your connection string into your application code

☐ Include full driver code example

```
mongodb+srv://zaq12345:<password>@cluster0.op2wo.mongodb.net/<dbname
```

Replace <password> with the password for the zaq12345 user. Replace <dbname> with the name of the database that connections will use by default. Ensure any option params are URL encoded.

Having trouble connecting? View our troubleshooting documentation

- router is defined in the following code before **app.listen(….**

```
const todoRouter = require('./routes/todos');

app.use('/todos', todoRouter);
```

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

- the following shows the router syntax…. But first add a directory called **routes** and add a file **todos.js**

```
const router = require("express").Router();
let Todo = require("../models/todo.model");

router.route("/").get((req, res) => {
  Todo.find()
    .then((todos) => res.json(todos))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/add").post((req, res) => {
  const activity = req.body.activity;
  const newTodo = new Todo({
    activity,
  });

  newTodo
    .save()
    .then(() => res.json("Todo added!"))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/:id").get((req, res) => {
  Todo.findById(req.params.id)
    .then((todo) => res.json(todo))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/:id").delete((req, res) => {
  Todo.findByIdAndDelete(req.params.id)
    .then(() => res.json("Todo deleted."))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/update/:id").post((req, res) => {
  Todo.findById(req.params.id)
    .then((todo) => {
      todo.activity = req.body.activity;

      todo
        .save()
        .then(() => res.json("Todo updated!"))
        .catch((err) => res.status(400).json("Error: " + err));
    })
    .catch((err) => res.status(400).json("Error: " + err));
});

module.exports = router;
```

- As can be seen we need to import the database schema
- Add the following code in **todo.model.js** after creating a **models** directory

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

const Schema = mongoose.Schema;

const todoSchema = new Schema({
  activity: { type: String, required: true },
});

const Todo = mongoose.model("Todo", todoSchema);
```

```
module.exports = Todo;
```

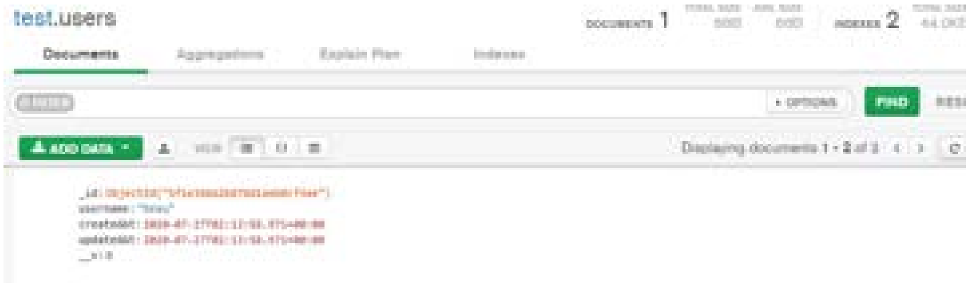
Server API Endpoints

- First let’s check the root route and /add route as defined in:

```
const router = require("express").Router();
let Todo = require("../models/todo.model");

router.route("/").get((req, res) => {
  Todo.find()
    .then((todos) => res.json(todos))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/add").post((req, res) => {
  const activity = req.body.activity;

  const newTodo = new Todo({
    activity,
  });

  newTodo
    .save()
    .then(() => res.json("Todo added!"))
    .catch((err) => res.status(400).json("Error: " + err));
});
```

- Let’s test ther server API through Postman for /, and /add
- You can also use compass to check the database… let’s first connect to our **uri** in compass
- go to postman and try post (make sure server is running):
  http://localhost:5000/todos/add with the following json data:
  { “activity” : “test1” }

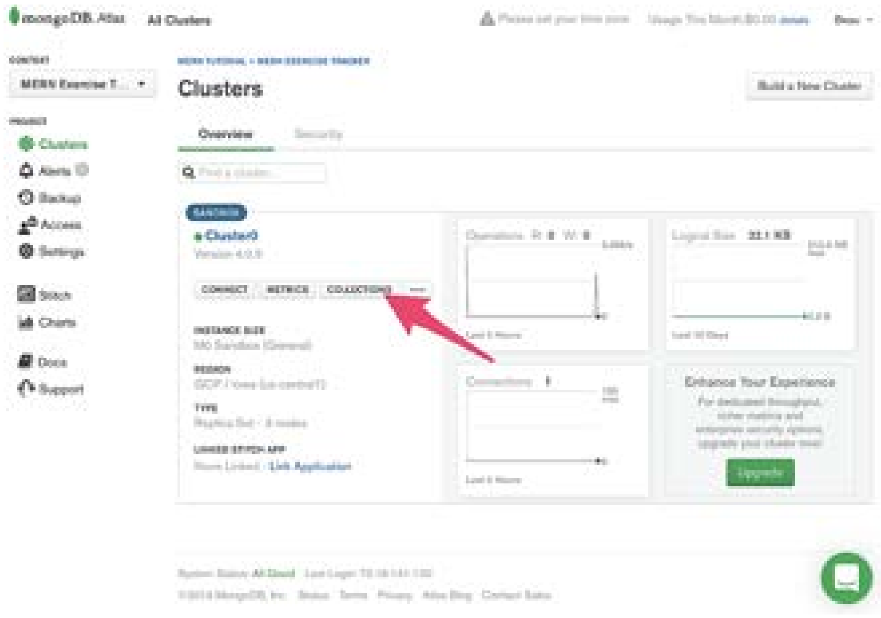
- Check in compass

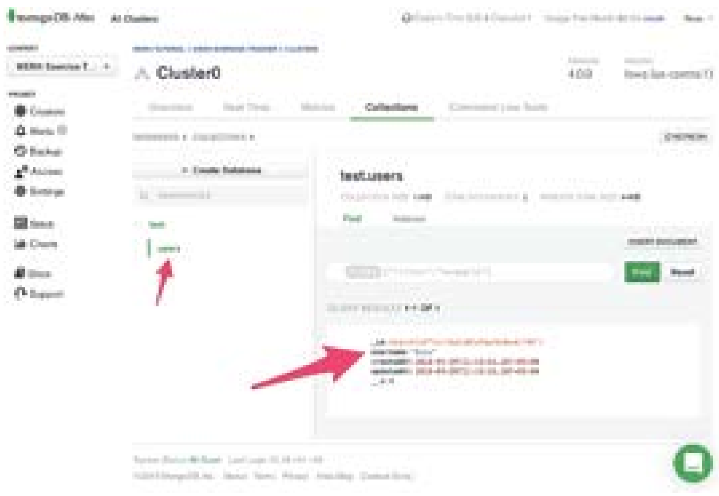


```
_id: [illegible]
username: [illegible]
createdAt: [illegible]
updatedAt: [illegible]
__v: 0
```

-

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





Let’s add a few exercises. Use Postman to POST the following data to “**http://localhost:5000/todos/add**” (update the username to the one you used).

- Add some data

```
{
"activity": "task2"
}
```

And also

```
{
"activity": "task3"
}
```

- For other CRUD operations, you can check the rest of the code.

```
router.route("/:id").get((req, res) => {
  Todo.findById(req.params.id)
    .then((todo) => res.json(todo))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/:id").delete((req, res) => {
  Todo.findByIdAndDelete(req.params.id)
    .then(() => res.json("Todo deleted."))
    .catch((err) => res.status(400).json("Error: " + err));
});

router.route("/update/:id").post((req, res) => {
  Todo.findById(req.params.id)
    .then((todo) => {
      todo.activity = req.body.activity;

      todo
        .save()
        .then(() => res.json("Todo 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 todo.save to save the updated object in the database.

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

- Now try GET in postman with id, http://localhost:5000/todos/5f1f7ff16348a030fb7d6677
- Try it with DELETE as well
- Now let's try UPDATE with id, http://localhost:5000/todos/update/5f1f7fdd6348a030fb7d6676
  ```
  {
  "activity": "new Task",
  }
  ```
- Now we are finished with the backend
- **BACKEND COMPLETE!** 😊

CREATING THE FRONTEND

After finishing the backend, now time is for creating the frontend. We use React to build a simple front end.

Let's first discuss the React Router (you can use `npm init react-app project-name or npx create-react-app project-name)`

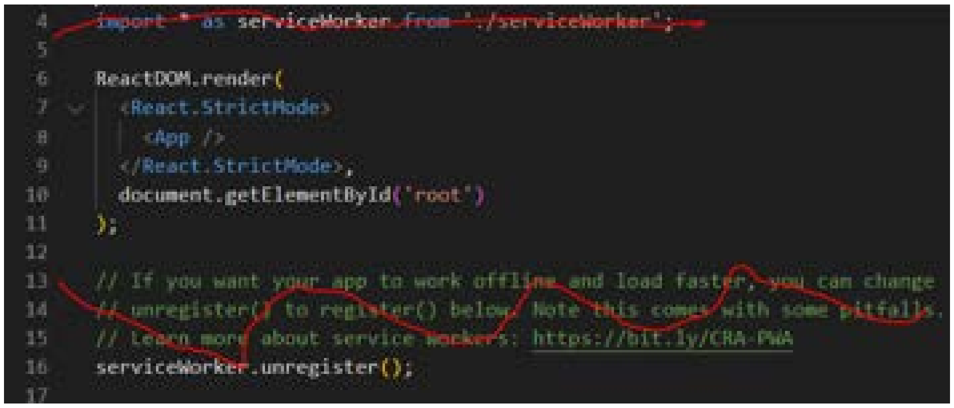
- Modify all unnecessary syntax and files.
  - Change the title

```
-       <title>Full Stack Practice</title>
```

  - Let's go to index.js
  - We can delete

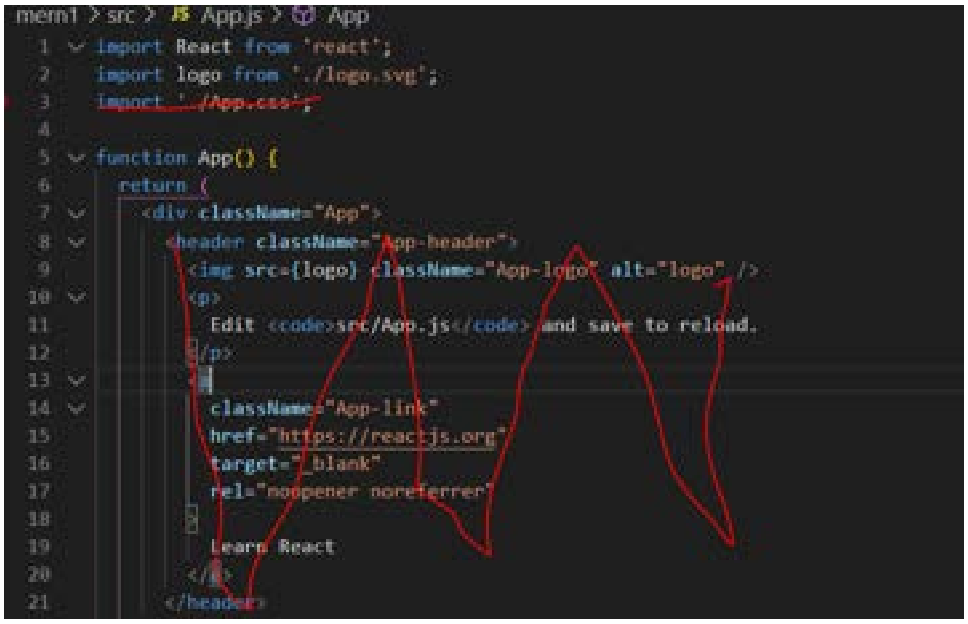
~~import './index.css';~~

  - We are going to make changes to App.js … As you see we are going to render App in the #root
  - Also get rid of the following



```
import * as serviceWorker from './serviceWorker';

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

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

  -
  - Let's go to App.js (the main app)
  - Let's go to **tracker** directory and **npm start**
  - In **App.js** delete below



```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
```

- **npm start** in the **tracker** directory
- we still need to add bootstrap module to our react so … **npm install bootstrap** and import it's CSS file in **App.js** by ..
- to start checking the front end… modify **App.js** as below and check if it renders

```
import React from 'react';
import "bootstrap/dist/css/bootstrap.min.css";

function App() {
 return (
   <div className="container">
     Hello World
   </div>
 );
}

export default App;
```

- also **npm install react-router-dom**

```
import React from "react";
import "bootstrap/dist/css/bootstrap.min.css";
import { BrowserRouter as Router, Route } from "react-router-dom";
```

the main **App.js** should contain the components that we want to have like below. The main components will be **<Navbar />**, **<TodosList />**, and **<CreateTodo />**:

```
import React from "react";
import "bootstrap/dist/css/bootstrap.min.css";
import { BrowserRouter as Router, Route } from "react-router-dom";

import Navbar from "./components/navbar.component";
import TodosList from "./components/todos-list.component";
import CreateTodo from "./components/create-todo.component";

function App() {
  return (
    <Router>
      <div className="container">
        <Navbar />
        <br />
        <Route path="/" exact component={TodosList} />
        <Route path="/create" component={CreateTodo} />
      </div>
    </Router>
  );
}
export default App;
```

- create **components** directory inside **src** directory
- create **create-todo.component.js**, **navbar.component.js**, **todos-list.component.js**
- we define the components in those files and import them as below

```
import Navbar from "./components/navbar.component";
import TodosList from "./components/todos-list.component";
import CreateTodo from "./components/create-todo.component";
```

- then create a file inside components called **navbar.component.js** and add the following code

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';

export default class Navbar extends Component {

  render() {
    return (
      <nav className="navbar navbar-dark bg-dark navbar-expand-lg">
        <Link to="/" className="navbar-brand">Todo List Maker</Link>
        <div className="collpase navbar-collapse">
        <ul className="navbar-nav mr-auto">
          <li className="navbar-item">
          <Link to="/" className="nav-link">Tasks</Link>
          </li>
          <li className="navbar-item">
          <Link to="/create" className="nav-link">Add A Task</Link>
          </li>
        </ul>
        </div>
      </nav>
    );
  }
}
```

- This is just the navbar from the Bootstrap documentation converted to work for our purposes.
- Next, in the components directory create the following files:
    - todos-list.component.js
    - create-todo-exercise.component.js

- We can create a simple component first to test if the router works and then add the following code

(Note: For the codes below , do not pay attention to **axios** module … it will be discussed later)

the code for these components are as follows:

- For **todos-list.component.js** as in the following:

```
import React, { Component } from 'react';
// import { Link } from 'react-router-dom';
import axios from 'axios';

const Todo = props => (
  <tr className="d-flex">
    <td  className='col-10'>{props.todo.activity}</td>
    <td className='col-2' style={{textAlign:"right"}}>
      <button onClick={() => { props.deleteTodo(props.todo._id) }} >delete</button>
    </td>
  </tr>
)

export default class TodosList extends Component {
  constructor(props) {
    super(props);

    this.deleteTodo = this.deleteTodo.bind(this)

    this.state = {todos: []};
  }

  componentDidMount() {
    // // this is for testing
    // this.setState({
    //   todos: [{activity:'t1'},{activity:'t2'},{activity:'t3'}]
    // })
    axios.get('http://localhost:5000/todos/')
      .then(response => {
        this.setState({ todos: response.data })
      })
      .catch((error) => {
        console.log(error);
      })
  }

  deleteTodo(id) {
    axios.delete('http://localhost:5000/todos/'+id)
      .then(response => { console.log(response.data)});

    this.setState({
      todos: this.state.todos.filter(el => el._id !== id)
    })
  }

  todoList() {
    return this.state.todos.map(currenttodo => {
      return <Todo todo={currenttodo}
      deleteTodo={this.deleteTodo} key={currenttodo._id}

      />;
    })
  }

  render() {
    return (
      <div>
        <h3>Logged Todos</h3>
        <table className="table">
          <thead className="thead-light">
            <tr>
              <th>Activity</th>
            </tr>
          </thead>
          <tbody>
            { this.todoList() }
          </tbody>
          {/* just to put the last line under */} <tbody><a> </a></tbody>
        </table>
      </div>
    )
  }
}
```

And for **create-todo.component.js** as below:

```
import React, { Component } from "react";
import axios from "axios";
// import DatePicker from 'react-datepicker';
// import "react-datepicker/dist/react-datepicker.css";

export default class CreateTodo extends Component {
  constructor(props) {
    super(props);

    this.onChangeActivity = this.onChangeActivity.bind(this);
    this.onSubmit = this.onSubmit.bind(this);

    this.state = {
      activity: "",
    };
  }

  onChangeActivity(e) {
    this.setState({
      activity: e.target.value,
    });
  }

  onSubmit(e) {
    e.preventDefault();

    const activityvar = {
      activity: this.state.activity,
    };

    console.log(activityvar);

    axios.post("http://localhost:5000/todos/add", activityvar).then((res) => {
      window.location = "/";
    });
  }

  render() {
    return (
      <div>
        <h3>Create New Task</h3>
        <form onSubmit={this.onSubmit}>
          <div className="form-group">
            <label>New Task: </label>
            <input
              type="text"
              required
              className="form-control"
              value={this.state.activity}
              onChange={this.onChangeActivity}
            />
          </div>

          <div className="form-group">
            <input
              type="submit"
              value="Create Activity Log"
              className="btn btn-primary"
            />
          </div>
        </form>
      </div>
    );
  }
}
```

- Axios module in the codes above is for connecting these React Frontend components to the backend.
    - In terminal type **npm I axios**

Connecting Front to Back

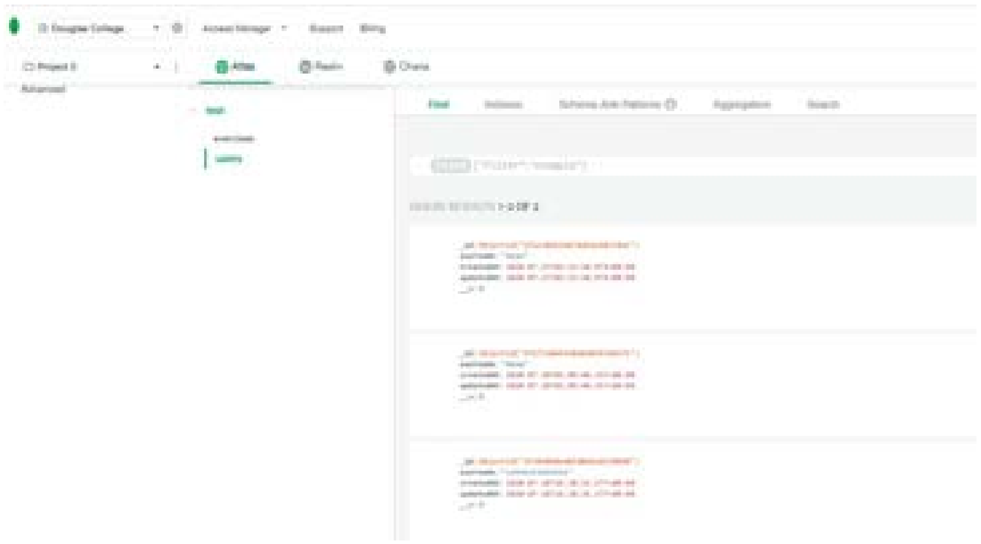
Let's connect the frontend to backend. We will have our frontend to send HTTP request to the server endpoints on the backend. You have already noticed we used a module called axios to perform the API call and get the data from the server. The following describes how we did it.

- Axios library is used to send HTTP requests to our backend.
- Make sure axios is installed and is imported in create-todo.component.js and todo-list.component.js
- 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("http://localhost:5000/todos/add", activityvar)
      .then((res) => console.log(res.data));
```

- Where **activityvar** is the property changed through the state
- The axios.post method sends an HTTP POST request to the backend endpoint http://localhost:5000/todos/add. This endpoint is expecting a JSON object in the request body so we passed in the newUser object as a second argument.

- We can add tasks and check in the backend
- Go to backend directory and start **nodemon server** also run the front end
- Let's add a new task
-



[illegible]

-
- Now go and give the program some example and check the results in Atlas