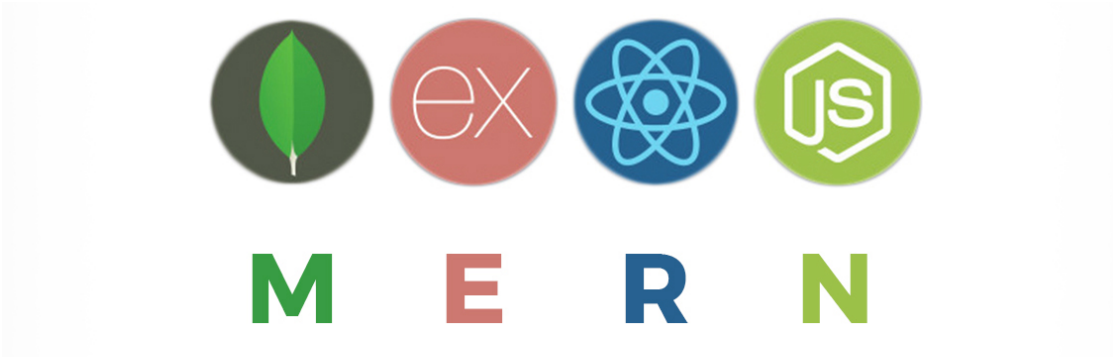
# MERN

## STACK DEVELOPMENT

A R I H A R A S U D H A N

# THE MERN STACK

A Stack is the various tiers or technologies. MERN is a great web developing stack. It is definitely a good choice for web application projects.




MERN stands for MongoDB, ExpressJS, ReactJS and NodeJS. There are other stacks of technologies used for web development such as LAMP, MEAN, ROR , FARM and so on. Let's learn the technologies in MERN Stack Development.

MongoDB : A NoSQL database used for persistent data storage

Node.js : A server-side JavaScript runtime environment

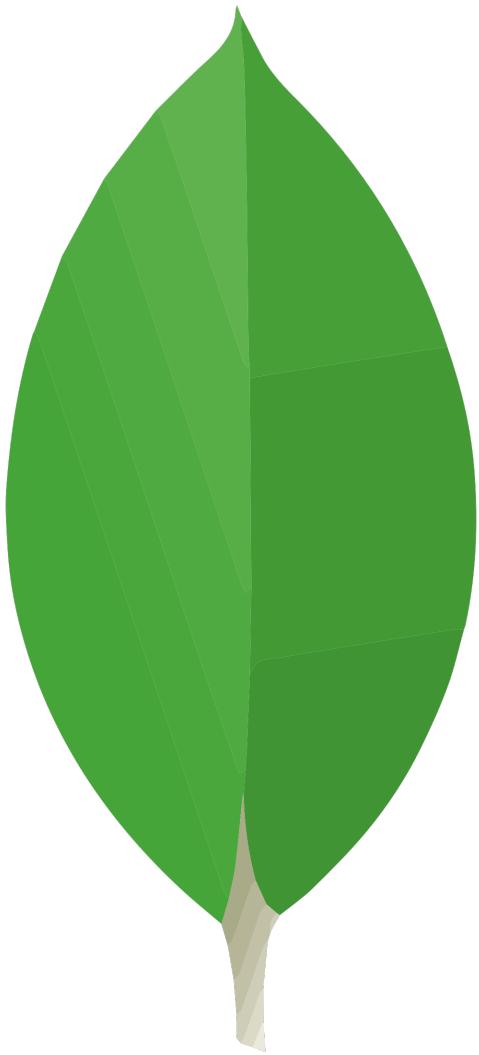
Express : A web server built on Node.js, forming the middle tier, or the web server.

React : A front-end technology from Facebook

# MongoDB

MongoDB is the database used in the MERN stack. It is a NoSQL document-oriented database, with a flexible schema and a JSON-based query language. NoSQL means “non-relational”. MongoDB is a document-oriented database. The unit of storage ( As row of a Relational Database ) is a document, or an object, and multiple documents are stored in collections ( As table of a Relational Database ). Every storage or a row has a unique identifier which is assigned automatically and by means of which the row can be accessed. For MongoDB, the query language is based on JSON. Data is also interchanged in JSON format. In fact, the data is natively stored in a variation of JSON called BSON (where B stands for Binary) in order to efficiently utilize space. When you retrieve a document from a collection, it is returned as a JSON object.

We can interact with MongoDB using a shell that comes with MongoDB. We can also write code in JavaScript to perform operations.



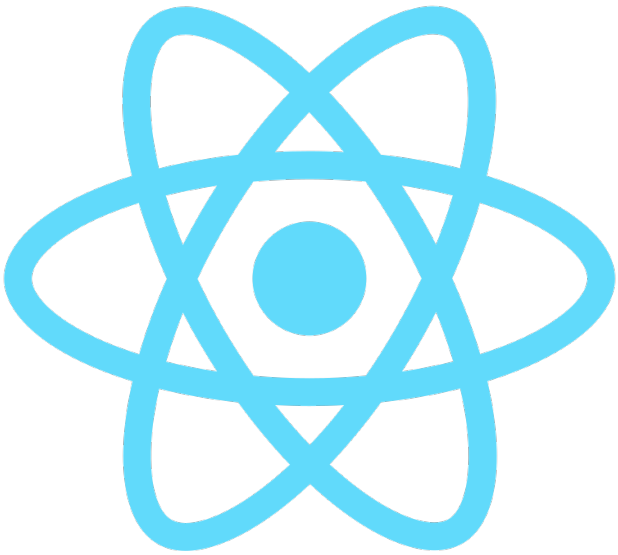
mongoDB

# Express

ExpressJS simplifies the task of writing Server Code as it is a bit hard process with mere Node JS. Express parses request URL, headers, and parameters for us. It does more for us. In essence, Express is a web server framework meant for Node.js, and it is not very different from many other server-side frameworks in terms of what we can achieve with it.

# REACT JS

React is a JavaScript frontend library. It was developed by FaceBook for Ads React was born not in the Facebook application that we all see, but rather in Facebook's Ads organization. In React, the template controls the state. React uses the so-called Virtual DOM Concept for handling the state changes. It is like a datastructure over the actual DOM. It looks for the change in a particular point and applies that to that particular part. It can be made clear when we learn the concept of components in React. Everything in React is a component.



# NODE JS

Node is simply , “JavaScript outside of a Browser”. It provides a platform for running JavaScript outside of a browser such as Server. There are many Node Modules we can use. Like The pip of Python, npm is a package manager of Node. It stands for node package manager. Node.js has an asynchronous, event-driven, non-blocking input/output (I/O) model, as opposed to using threads to achieve multitasking. Node.js achieves multitasking using an event loop. This is nothing but a queue of events that need to be processed and callbacks to be run on those events.

# HELLO MAKKAA !

In the Hello Makkaa exercise, we'll use React to render a simple page and use Node.js and Express to serve that page from a web server. Let's create a simple serverless page.

```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="UTF-8" />
  <title>MERN Stack - ARI</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">
</script>
</head>
<body>
  <div id="here"></div>
  <script type="text/babel">
     var contentNode = document.getElementById('here');
     var component = <h1>Hello Makkaa!</h1>;
     ReactDOM.render(component, contentNode);
  </script>
</body>
</html>
```

As we observe, we have given three Content Delivery Network Links in the head which we need for using ReactJS. Then, we have created an empty node with id of 'here'.

The script tag is specified with the babel compiler which indeed is the compiler for the React Code which we call as JSX. Inside the script tag, we write some JSX Codes. It may feel like strange to call it JSX. JSX is a combination of HTML and JavaScript. We take the empty node. Then, we create a component (Just think it as a division for now). Using ReactDOM.render() method, we render the component inside the empty contentNode. Open it in a browser.


**Hello Makkaa!**

Figure 1: First Program Output

The JSX code gets transformed into JavaScript code in background. After transformation, this is what the code that is generated will look like :

```
var component = React.createElement('h1', null, 'Hello Makkaa!');
```

# SERVER

It is a good practice to use the node version manager. Using nvm, install node as the following.

**> nvm install node**

If you want to install a particular version, you can use the following command,

**> nvm install 4.7**

Once you have installed NodeJs, you can check the version by,

**> node - - version**

If you see the version, that's it! You have installed it successfully. Before installing any third party package, it's good to initialize a project using the npm init command. nvm stands for node version manager. Meanwhile, npm stands for node package manager. Create a directory like mymern, get into that using cd and initialize the project.

**> mkdir mymern**

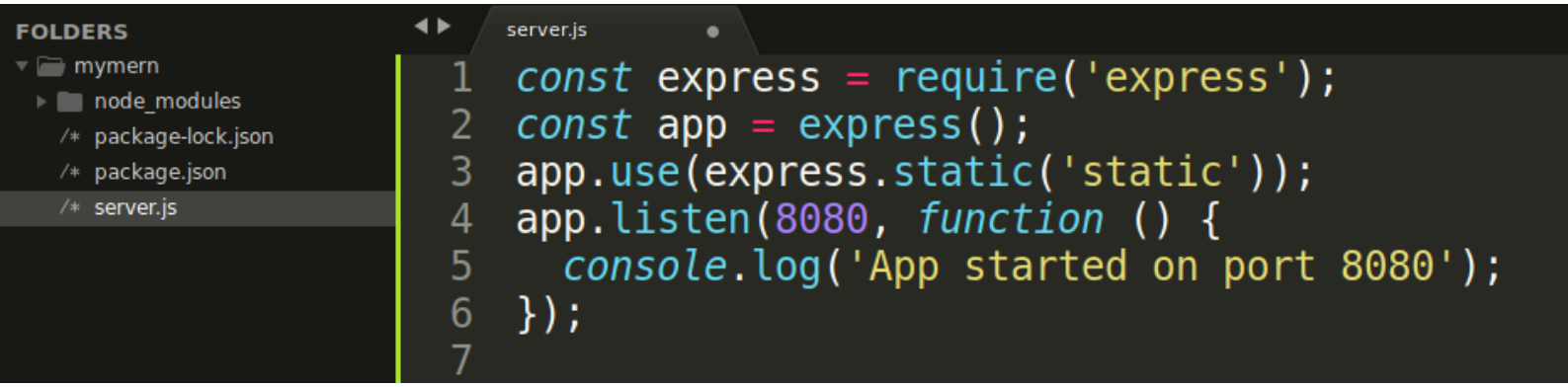
**> cd mymern**

**> npm init**

Now, let's install express. To install any libraries, we have to use npm install command.

**> npm install express**

After installing these all, let's create a javascript file inside the mymern folder. It is going to be our server.




What's there in these lines? The first line just imports the express library. We create the server app in the second line. The express.static function responds to a request by trying to match the request URL with a file under a directory specified by the parameter to the generator function. Just ignore it if you don't get it for now. The magic is in the next line.

The listen() method takes in a port number and starts the server, which then waits for requests like the genie out of the lamp. ( Maybe the casper genie )

So, how can we run it? Go to the terminal or cmd and type ,

**> npm start**

Man! It will show you where (Talking of the port) the server is waiting for the requests.

```
(base) ari-pt7127@ariharasudhan:~/Desktop/mymern$ npm start

> mymern@1.0.0 start
> node server.js

App started on port 8080
```

# THE REACT PURANAM

Kindly visit the following link to get well versed in ReactJS :
arihara-sudhan.github.io/articles.html

React is an Open Source library created by Facebook. It helps for UI/UX Design of modern web applications. React uses JSX which allows us to write HTML directly within JavaScript.

## JSX

JSX is similar to the HTML. We can write JavaScript directly within JSX. To do this, simply include the JavaScript code within curly braces: { ‘JavaScript code’ }. JSX code must be compiled into JavaScript. Babel is the transpiler used for this.

ReactDOM.render(JSX,document.getElementById(‘root’)). This function call is what places your JSX into React’s own lightweight representation of the DOM. React then uses snapshots of its own DOM to optimize updating only specific parts of the actual DOM.

```
const JSX = <p>Hello World</p>;
```

To write a complex JSX element, there must be a top-most element. Say, 

```
const JSX = <div>
    <h1> Hello </h1>
    <p> Hiii </p>
    <ul>
        <li>Ari</li>
        <li>Hara</li>
        <li>Sudhan</li>
   </ul>
   </div> ;
```

Comments are non-executable statments,used for describing what happens in the code or for just avoiding a statement.

```
const JSX = (
  <div>
      {/*I am a Comment*/}
  </div>
);
```

Do we find paranthesis? Using them is a good practice ( for grouping ).

## CDNs

Ok. Now, we are aware of how JSX syntax looks like. Add these following CDNs to your document.

```
<head>
  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
  <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
```

## Rendering JSX

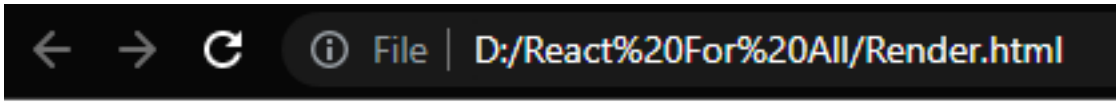
Let’s render our JSX to the DOM. ReactDOM offers a simple method to render React elements to the DOM which looks like this: ReactDOM.render(whatToBeRendered, whereToBeRendered). Hmm… We have to determine where to render. Let’s create a division with an ID of here.

```
<body>
  <div id="mydiv"></div>
</body>
```

Now, the script element must have a type of text/babel which is our transpiler discussed earlier.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    const JSX = <h1>I am ARI</h1>;
    ReactDOM.render(JSX, document.getElementById('here'));
  </script>
</body>
```

Our output would be,



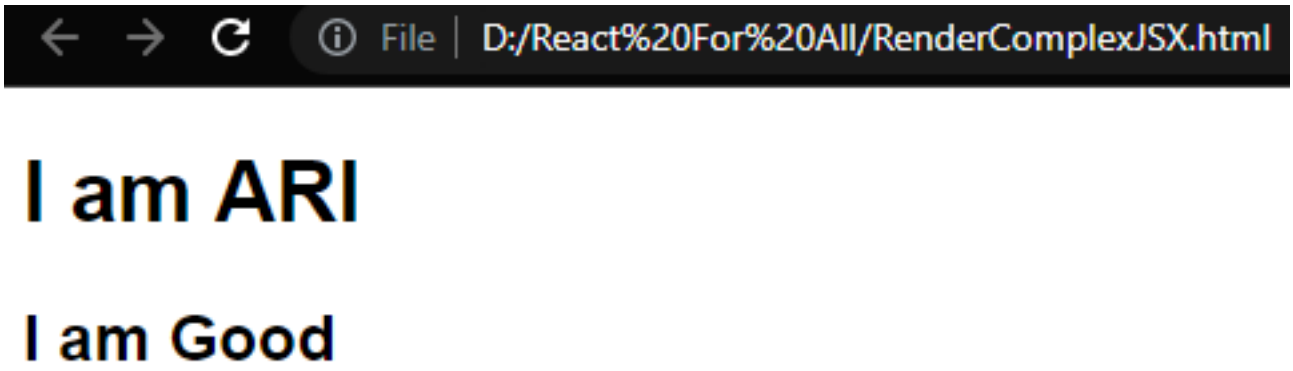

# I am ARI

We can do it for a complex JSX Element. But, never forget the Top-Most element. We can use  or <></>.

I mean,

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    const JSX = (<div>
                  <h1>I am ARI</h1>
                  <h2>I am Good</h2>
                 </div>);
    ReactDOM.render(JSX, document.getElementById('here'));
  </script>
</body>
```

The output is,



## ClassName

To add a class to the JSX Element, the term used is className and not class as in HTML. The class is a reserved word in JavaScript. Let's add a className of "myClass" to our JSX.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
  const JSX = (
    <div className="myClass">
      <h1>Vana <br /> kkam</h1>
      <hr />
    </div>
  );
      ReactDOM.render(JSX, document.getElementById('here'));
  </script>
</body>
```

## Self Closing Tags

JSX differs from HTML in some ways like className vs. class for defining HTML classes. Another important way in which JSX differs from HTML is in the concept self-closing tag. In HTML, almost all tags have both an opening and closing tag: <h1></h1>;

; The self-closing tags don't require both an opening and closing tag.

For an instance, the line-break tag can be written as <br> or as <br />, but should never be written as <br></br>.

In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. The line-break tag must always be written as <br />.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
  const JSX = (
    <div className="myClass">
      <h1>Vana <br /> kkam</h1>
      <hr />
    </div>
  );
      ReactDOM.render(JSX, document.getElementById('here'));
  </script>
</body>
```


**Vana**
**kkam**

---

## Props

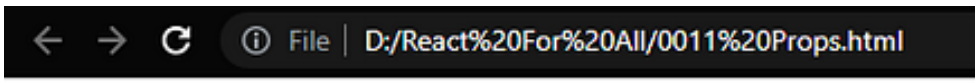
In React, we can pass props, or properties, to child components.

<Ari name=’Ariharasudhan’ />

In the above line, we write the component Ari and pass the string value name of ‘Ariharasudhan’ to the props the component Ari.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
      const Ari = (props) => {
          return <h1>Hi {props.name}</h1>;
      };
      const Func = () =>{
          return (
            <div>
                <Ari name='Ariharasudhan'/>
            </div>
          );
      }
      ReactDOM.render(<Func />, document.getElementById('here'));
  </script>
</body>
```

To get the value passed to the props, { props.var_name } is the syntax as shown above.




# Hi Ariharasudhan

We can also pass an array as an argument to props.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    const MyList = (props) => {
      return <h1>{props.works.join(', ')}</h1>
    };

    class MyWorks extends React.Component {
      constructor(props) {
        super(props);
      }
      render() {
        return (
          <div>
            <h1>To Do Lists</h1>
            <h2>Today</h2>
            <MyList works={["learning", "playing"]}/>
            <h2>Tomorrow</h2>
            <MyList works={["learning", "playing","repeating"]}/>
          </div>
        );
      }};
      ReactDOM.render(<MyWorks />, document.getElementById('here'));
  </script>
</body>
```


# To Do Lists

## Today

## learning, playing

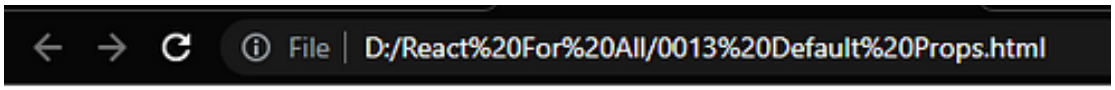
## Tomorrow

## learning, playing, repeating

## Default Props

React also has an option to set default props. We can define the default props as below.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    const MyName = (props) => {
      return <h1>Hello {props.name}</h1>
    };
    MyName.defaultProps = {name:'Ariharasudhan'};
    ReactDOM.render(<MyName />, document.getElementById('here'));
  </script>
</body>
```




# Hello Ariharasudhan

If we render the MyName component with our own element say, <MyName name='Ari'/>, it means we override the default component.

## PropTypes

React provides useful type-checking features to verify appropriate types.

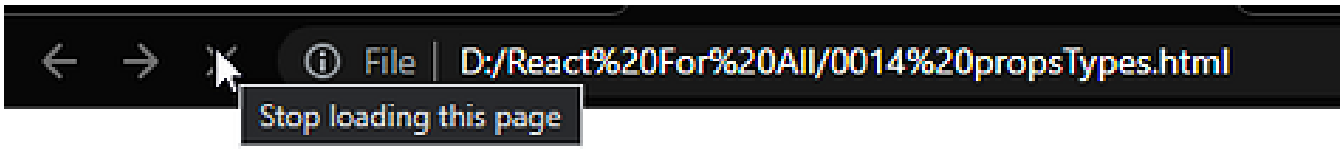
It's considered a best practice to set propTypes when we are aware of the type of a prop . propTypes property can be defined for a component in the same way the defaultProps are defined. Doing this will let us go with appropriate types. Before using propTypes, we need another CDN for this.

The Concept is,

```
<body>
  <div id="here"></div>
  <script type="text/babel">
      const MyComponent = (props) => {
              return <h1>{props.name} is {props.age} years old</h1> };

      MyComponent.defaultProps = {
        age: 19,
        name: 'Ariharasudhan'
      };

      MyComponent.propTypes = {
        age: PropTypes.number.isRequired,
        name: PropTypes.string.isRequired
      }
      ReactDOM.render(<MyComponent />, document.getElementById('here'));
  </script>
</body>
```



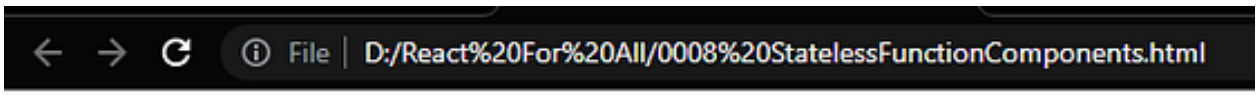

**Ariharasudhan is 19 years old**

# COMPONENTS THE ALL

Everything in React is a component. There are two ways to create a React component. The first way is to use a JavaScript function.A stateless component is the one that can receive data and render it, but does not manage that data.

To create a component with a function, simply write a JavaScript function that returns either JSX or null.For an instance,

```
<body>
  <div id="here"></div>
  <script type="text/babel">
  const MyCompo = function(){
      return(
      <div className="myClass">
        <h1>Components</h1>
        <h1>in react js</h1>
      </div>);
    };
    ReactDOM.render(<MyCompo />, document.getElementById('here'));
  </script>
</body>
```




## Components

## in react js

Another way to create components is using classes as shown below.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
      class Ari extends React.Component {
        constructor(props) {
          super(props);
        }
        render() {
          return (
            <h1>Hello Makkale</h1>
          );
        }
      }
      ReactDOM.render(<MyCompo />, document.getElementById('here'));
  </script>
</body>
```


## Hello Makkale

This creates a class Ari which extends the React.Component class. So the Ari class now has access to many useful React features, such as local state and lifecycle hooks. No need to worry if we aren't familiar with these yet. The constructor uses super() to call the constructor of the parent class, in this case React.Component. The constructor is a special method used during the initialization of objects that are created with the class keyword. It is best practice to call a component's constructor with super, and pass props to both. This makes sure the component is initialized properly. Ari is defined in the code editor using class syntax. Finish writing the render method so it returns what to be rendered. WE MUST HAVE A RENDER( ) THAT RETURNS!

## Composing Components

To compose the components together, we gotta create a parent component which renders each of these components as children.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
      const Compo = () => {
          return <h1>Hi</h1>;
      };
      class Ari extends React.Component {
        constructor(props) {
          super(props);
        }
        render() {
          return (
            <div>
                <h1>I am ARI</h1>
                <Compo />
            </div>
          );
        }
      }
      ReactDOM.render(<Ari />, document.getElementById('here'));
  </script>
</body>
```


**I am ARI**

**Hi**

## The this Keyword

To refer a class component within itself, we have to use the this keyword. To access a props within a component, we have to use the syntax of { this.props.data }

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    class MyClassCompo extends React.Component {
      constructor(props){
        super(props)
    }
      render() {
        return <h1>Welcome {this.props.name}!</h1>
      }
    }

    ReactDOM.render(<MyClassCompo name="Ari"/>, document.getElementById('here'));
  </script>
</body>
```


**Welcome Ari!**

## Stateless Components

A stateless functional component is any function which accepts props and returns JSX. A stateless component is a class that extends React.Component, but does not use internal state.

Here we have a stateless component.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    const StatelessCompo = (props) =>{
      return <h1>Vanakkam {props.name}</h1>
    }

    ReactDOM.render(<StatelessCompo name="Ari"/>, document.getElementById('here'));
  </script>
</body>
```


**Vanakkam Ari**

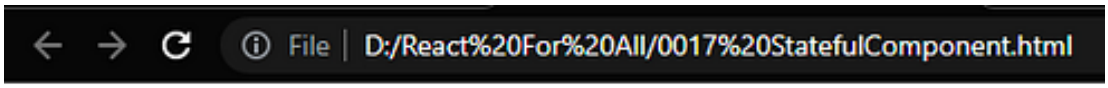
## Stateful Components

A stateful component is a class component that maintains its own internal state. State consists of any data, that can change over time.

A state can be created within the constructor. This initializes the component with state when it is created. The state must be set to a JS object as below :

this.state = { }

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    class StateFullCompo extends React.Component {
      constructor(props){
        super(props);
        this.state = {
          name: 'Ariharasudhan'
        }
      }
      render(){
        return <h1>Hello {this.state.name} !</h1>
      }
    };
    ReactDOM.render(<StateFullCompo />, document.getElementById('here'));
  </script>
</body>
```



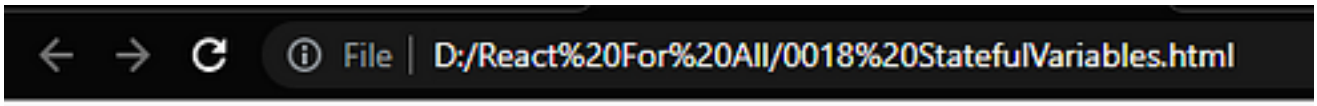

**Hello Ariharasudhan !**

If we make a component stateful, no other components are aware of its state. Its state is completely encapsulated.

## A State Value can be assigned

In the render() method, before the return statement, we can write JavaScript. we can declare functions, access data from state or props, perform computations on this data, and so on.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    class StateFullCompo extends React.Component {
      constructor(props){
        super(props);
        this.state = {
          name: 'Ariharasudhan'
        }
      }
      render(){
        const name = this.state.name;
        return <h1>Hello {name} !</h1>
      }
    };
    ReactDOM.render(<StateFullCompo />, document.getElementById('here'));
  </script>
</body>
```




**Hello Ariharasudhan !**

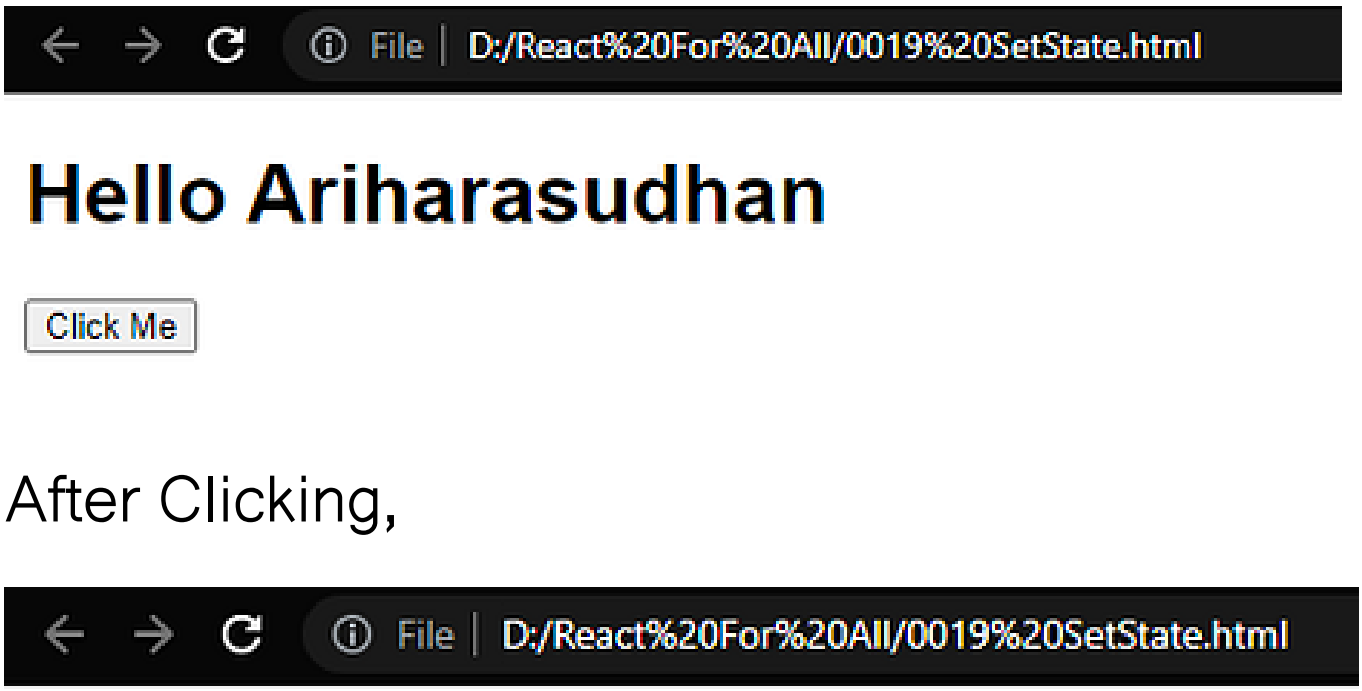
## SetState : Changing State Values

React has a method for updating component state called setState.Syntax to call it is ,

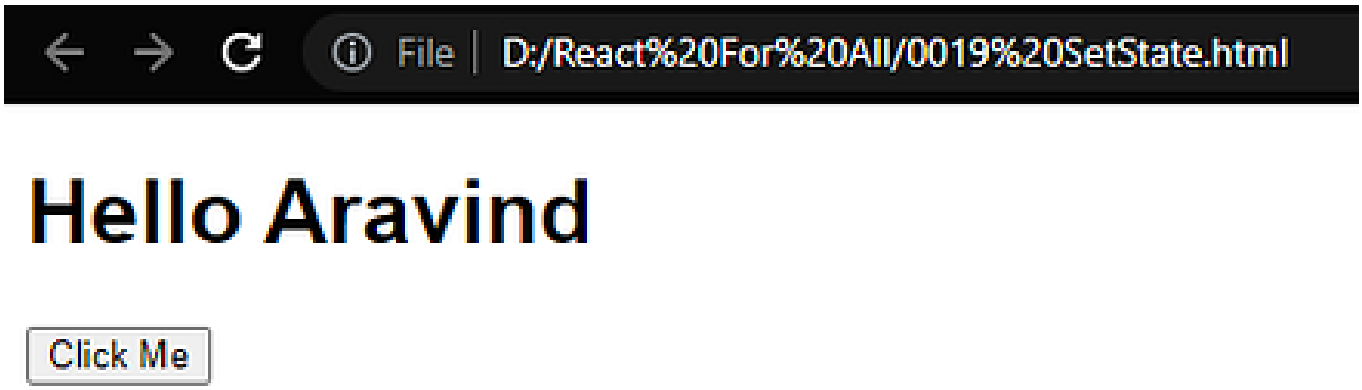
this.setState( { Key: Value } );

The keys are your state properties and the values are the updated state data. Let's create a button on clicking which the state gets updated.

```
<body>
  <div id="here"></div>
  <script type="text/babel">
    class StateFullCompo extends React.Component {
      constructor(props){
        super(props);
        this.state = {
          name: 'Ariharasudhan'
        }
        this.clickHandler = this.clickHandler.bind(this);
      }
      clickHandler() {
        this.setState( { name: 'Aravind' } );
      }
      render(){
        const name = this.state.name;
        return (
          <div>
            <h1> Hello {name} </h1>
            <button onClick={this.clickHandler}> Click Me </button>
          </div>
        );
      }
    };
    ReactDOM.render(<StateFullCompo />, document.getElementById('here'));
  </script>
</body>
```




After Clicking,



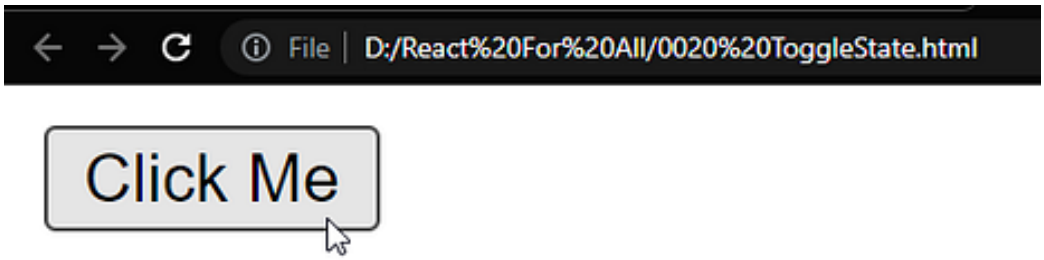
## Toggle Between States

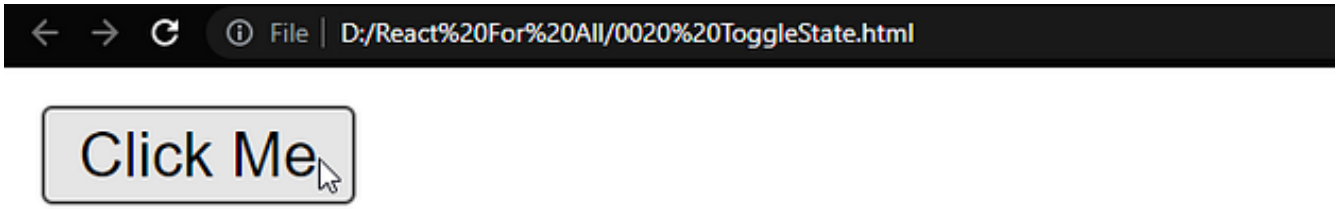
Sometimes you might need to know the previous state when updating the state.

```
constructor(props) {
  super(props);
  this.state = {
    visibility: false
  };
  this.toggleVisibility = this.toggleVisibility.bind(this);
}
toggleVisibility() {
  this.setState(state => {
    if (state.visibility === true) {
      return { visibility: false };
    } else {
      return { visibility: true };
    }
  });
}
```

The way followed here to toggle between states is so simple! We have a toggleVisibility method that comprises the setState method and takes state as an argument. It checks the argument.variable value and does what has to be done.

If we call this method onClicking a button, here what we got are,




## Let's design a counter

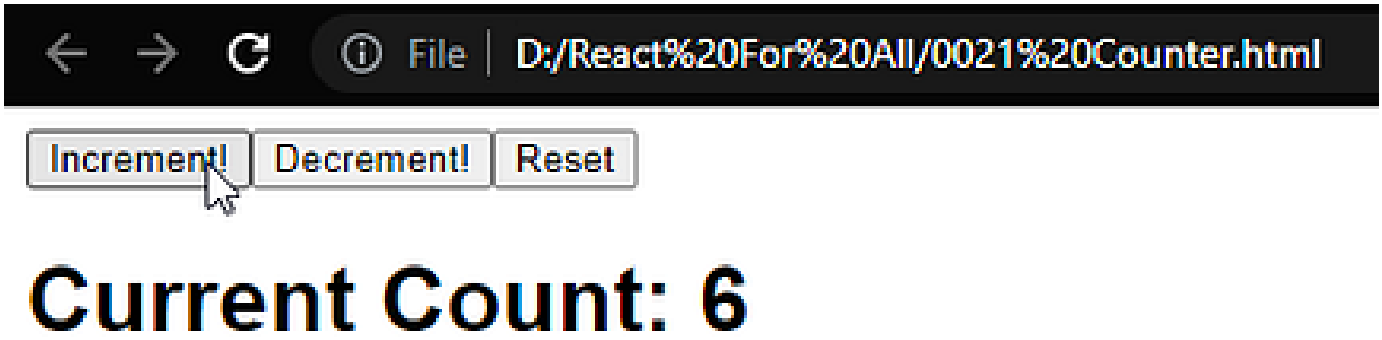
The Counter component keeps track of a count value in state. There are two buttons which call methods increment() and decrement(). Write these methods so the counter value is incremented or decremented by 1 when the appropriate button is clicked.

Also, create a reset() method so when the reset button is clicked, the count is set to 0.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  this.increment = this.increment.bind(this);
  this.decrement = this.decrement.bind(this);
  this.reset = this.reset.bind(this);
 }
  reset() {
    this.setState({
      count: 0
    });
  }
  increment() {
    this.setState(state => ({
      count: state.count + 1
    }));
  }
  decrement() {
    this.setState(state => ({
      count: state.count - 1
    }));
  }
```

```
    render() {
      return (
        <div>
          <button onClick={this.increment}>Increment!</button>
          <button onClick={this.decrement}>Decrement!</button>
          <button onClick={this.reset}>Reset</button>
          <h1>Current Count: {this.state.count}</h1>
        </div>
      );
    }
  }
  ReactDOM.render(<Counter/>, document.getElementById('here'));
```

The output is,




## State as Props

State can be passed as props. Consider a component named College. It has a state of two state members such as student and clgname with the values of “Ariharasudhan” and “Einstein College of Engineering” respectively.

It renders the another component Student. Besides, we have passed the clgname as props to the Student component.

```
class College extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      student: 'Ariharasudhan',
      clgname: 'Einstein College of Engineering'
    }
  }
  render() {
    return (
        <div>
          <Student clgname={this.state.clgname}/>
        </div>
    );
  }
};
```

```
class Student extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
    <div>
      <h1>I am from {this.props.clgname}</h1>
    </div>
    );
  }
};
ReactDOM.render(<College/>, document.getElementById('here'));
```

The output is,

## Multiple State Values as Props

We can pass many state values as props as shown below.

```
class College extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      student: 'Ariharasudhan',
      clgname: 'Einstein College of Engineering'
    }
  }
  render() {
    return (
        <div>
          <Student clgname={this.state.clgname} student={this.state.student}/>
        </div>
    );
  }
};
```

```
class Student extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
    <div>
      <h1>{this.props.student} is from {this.props.clgname}</h1>
    </div>
    );
  }
};
ReactDOM.render(<College/>, document.getElementById('here'));
```

Output is,


Remember to use the keyword this and curly braces { }. Okay... I think it's enough for now with ReactJS. Let's focus on other guys.

# REST APIs with Express

REST is a good pattern to adopt because it is simple and has very few constructs for building APIs.

| | | | | |
|---|---|---|---|---|
| Read – List | GET | Collection | GET /customers | Lists objects (additional query string can be used to filter) |
| Read | GET | Object | GET /customers/1234 | Returns a single object (query string may be used to filter fields) |
| Create | POST | Collection | POST /customers | Creates an object, and the object is supplied in the body. |
| Update | PUT | Object | PUT /customers/1234 | Replaces the object with the object supplied in the body. |
| Update | PATCH | Object | PATCH /customers/1234 | Modifies some attributes of the object, specification in the body. |
| Delete | DELETE | Object | DELETE /customers/1234 | Deletes the object |

The above listed are the CRUD ( Create, Read, Update, Delete ) operations and associated HTTP Verbs. Routing is a very very important operation. Request received is matched with the appropriate route and served with appropriate content ( response ). The code / function executed for a particular request is called handler code or the intended code. We can use the HTTP method by just calling the appropriate methods inside the express app. You can use the PostMan for API Checkup. In Visual Studio, we have an extension called REST CLIENT. We can use that too.

JS server.js > ...

```
1   const express = require('express');
2   const app = express();
3
4   app.get("/hi",(req,res)=>{
5       res.send("Hiii Makkaa")
6   })
7
8   app.listen(8080, function () {
9       console.log('App started on port 8080');
10  });
```

app.get() says that we are about to perform READ operation right now. It is actually a router for GET operation while we go to localhost:8080/hi. The second parameter in this method is a handler. For now, it simply sends a response of ‘Hii Makkaa’. Now, run ther server.js using node command and wake up the genie. Go to the REST Client / Postman and make a GET Request.

```
rest.REST > ...
    Send Request
1   GET http://localhost:8080/hi
2
```

Response :

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Express
3  Content-Type: text/html; charset=utf-8
4  Content-Length: 11
5  ETag: W/"b-UUsgTnYeW4HgxQwHCtcztN0vScw"
6  Date: Tue, 04 Apr 2023 05:19:28 GMT
7  Connection: close
8
9  Hiii Makkaa
```

Routing can be dynamic. For an instance, let's consider the following.

server.js > ...

```
1   const express = require('express');
2   const app = express();
3
4   app.get("/hi/:age",(req,res)=>{
5       res.send(`Hiii Makkaa! You are ${req.params.age} years old`)
6   })
7
```

The example above uses the :age which can be accessed in as req.params.age. It can be used for dynamic routing purposes.

rest.REST > ...

```
    Send Request
1   GET http://localhost:8080/hi/20
2
```

```
1   HTTP/1.1 200 OK
2   X-Powered-By: Express
3   Content-Type: text/html; charset=utf-8
4   Content-Length: 33
5   ETag: W/"21-/6emAjY5INVvyBjN1M5CrN0Z8to"
6   Date: Tue, 04 Apr 2023 05:25:51 GMT
7   Connection: close
8
9   Hiii Makkaa! You are 20 years old
```

The request and response objects are specified in the handler function. These objects contain some useful stuffs. For example : req.body, req.url, req.query , res.send, res.status, res.sendFile and so on. We can make it so interesting. Let's create an app that can perform CREATE and READ. Using npm command, let's install nodemon and body-parser which are for automatic restart of the server app and JSON handling respectively.After installing nodemon, we have to setup the package.json file so that we can start the server with a customized command. If we type npm start, it will execute nodemon server.js for us.

```
"scripts": {
  "start": "nodemon server.js"
},
```

Using nodemon will make our work a bit time saving since we don't need to restart the server each time manually.

Make sure to import the body-parser and configure for JSON Handling as shown below.

server.js > makkal

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
app.use(bodyParser.json())
```

Let's create an array of makkal which stores some details of people.

```
var makkal = [
    {   id: 7,
        name: "Ari",
        native: "Nellai",
        age: "20"
    },
    {
        id: 77,
        name: 'Aravind',
        native: "Tirunelveli",
        age: "20"
    }
]
```

Now, let's create a GET method for reading all the array items.

```
app.get("/read",(req,res)=>{
    res.send(makkal)
})
```

To create a new item for the array, let's create a POST method.

```
app.post("/create",(req,res)=>{
    makkal.push(req.body);
    res.send(makkal);
})
```

Don't forget to make the GENIE listen to us through a PORT.

```
app.listen(8080, function () {
    console.log('App started on port 8080');
});
```

Start the app using npm start command. Let's check out what happens.

When we invoke the GET request,

```
Send Request
GET http://localhost:8080/read/
```


```
[
  {
    "id": 7,
    "name": "Ari",
    "native": "Nellai",
    "age": "20"
  },
  {
    "id": 77,
    "name": "Aravind",
    "native": "Tirunelveli",
    "age": "20"
  }
]
```


When we invoke the POST request,

```
Send Request
POST http://localhost:8080/create
Content-Type: application/json

{
    "id": 777,
    "name": "Ariharasudhan",
    "native": "Tirunelveli",
    "age": "21"
}
```

Our new item is in.

```
[
  {
    "id": 7,
    "name": "Ari",
    "native": "Nellai",
    "age": "20"
  },
  {
    "id": 77,
    "name": "Aravind",
    "native": "Tirunelveli",
    "age": "20"
  },
  {
    "id": 777,
    "name": "Ariharasudhan",
    "native": "Tirunelveli",
    "age": "21"
  }
]
```

BOOM ! Let’s become a MERN Stack dev right below.

# With MongoDB

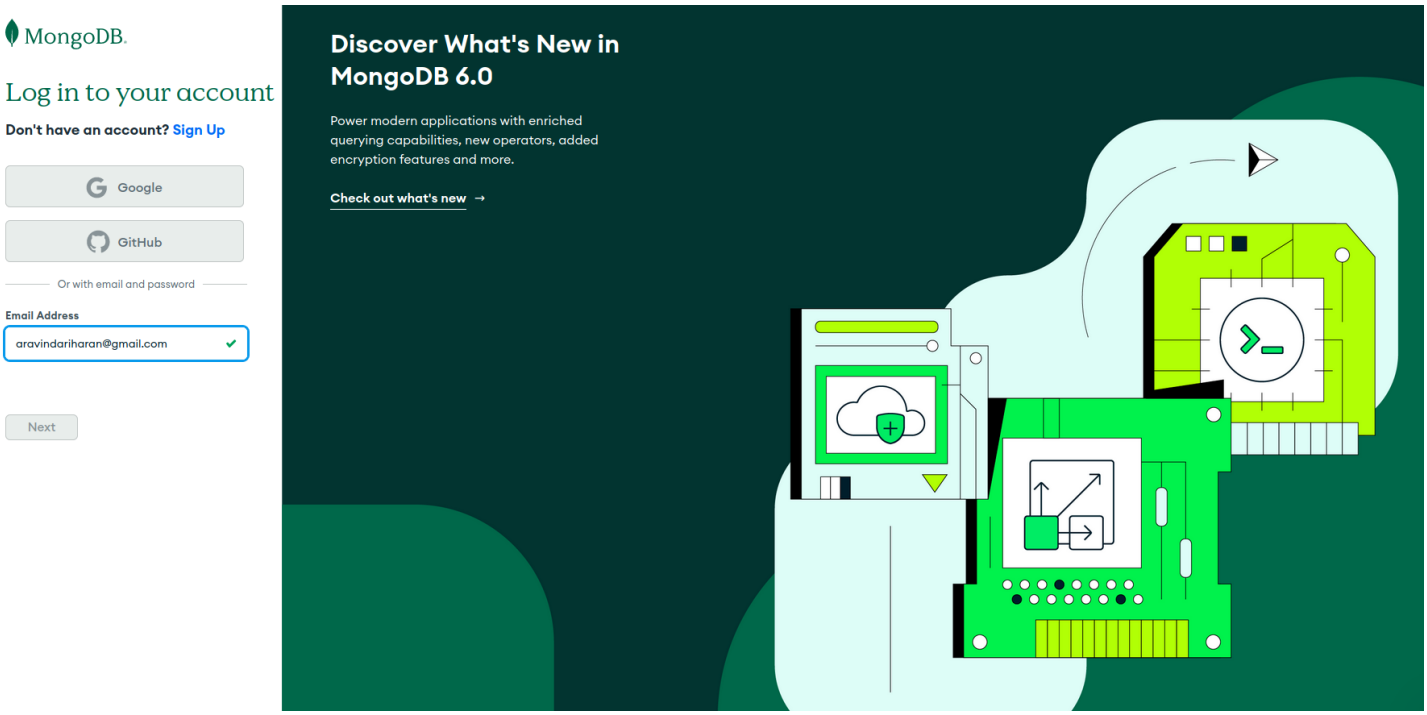
Let’s get familiar with the terminologies used int MongoDB.

DOCUMENT : Equivalent to a Row or a Record in Relational Database. MongoDB is a document database where an object has to be stored in document form with field-value pairs. It is similar to the JSON.

COLLECTION : Equivalent to a Table in a relational database. It is a set of documents.Even if you don’t supply an _id field when creating a document, MongoDB creates this field and auto-generates a unique key for every document.

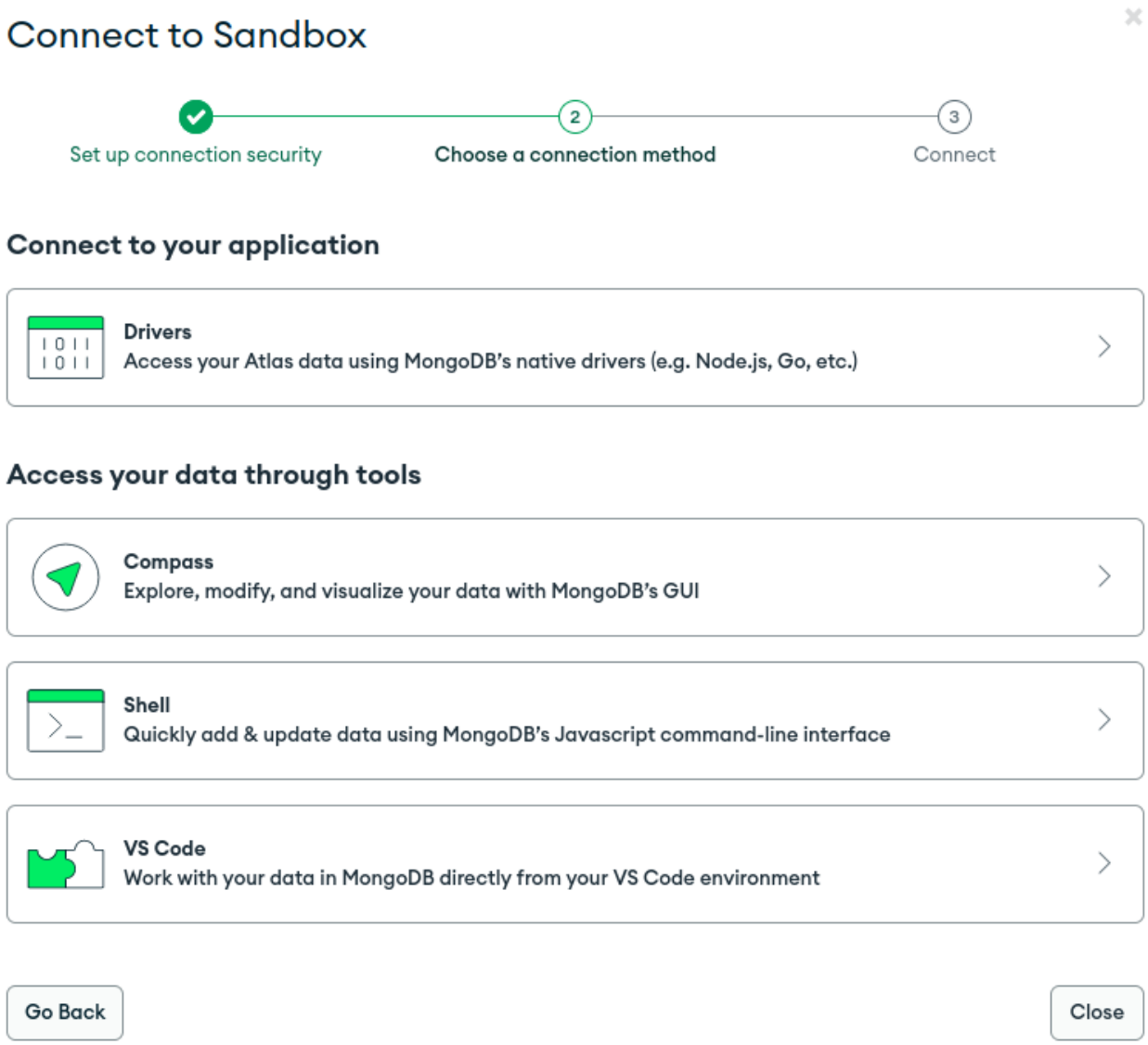
QUERY LANGUAGE : MongoDB query language is made up of different methods to achieve CRUD Operations. MongoDB encourages denormalization, that is, storing related parts of a document as embedded subdocuments rather than as separate collections (tables) in a relational database.

It is necessary to go and create a MongoDB account. After you create your account, Create your database and collection(s). Then, connect to the database using the connection string.

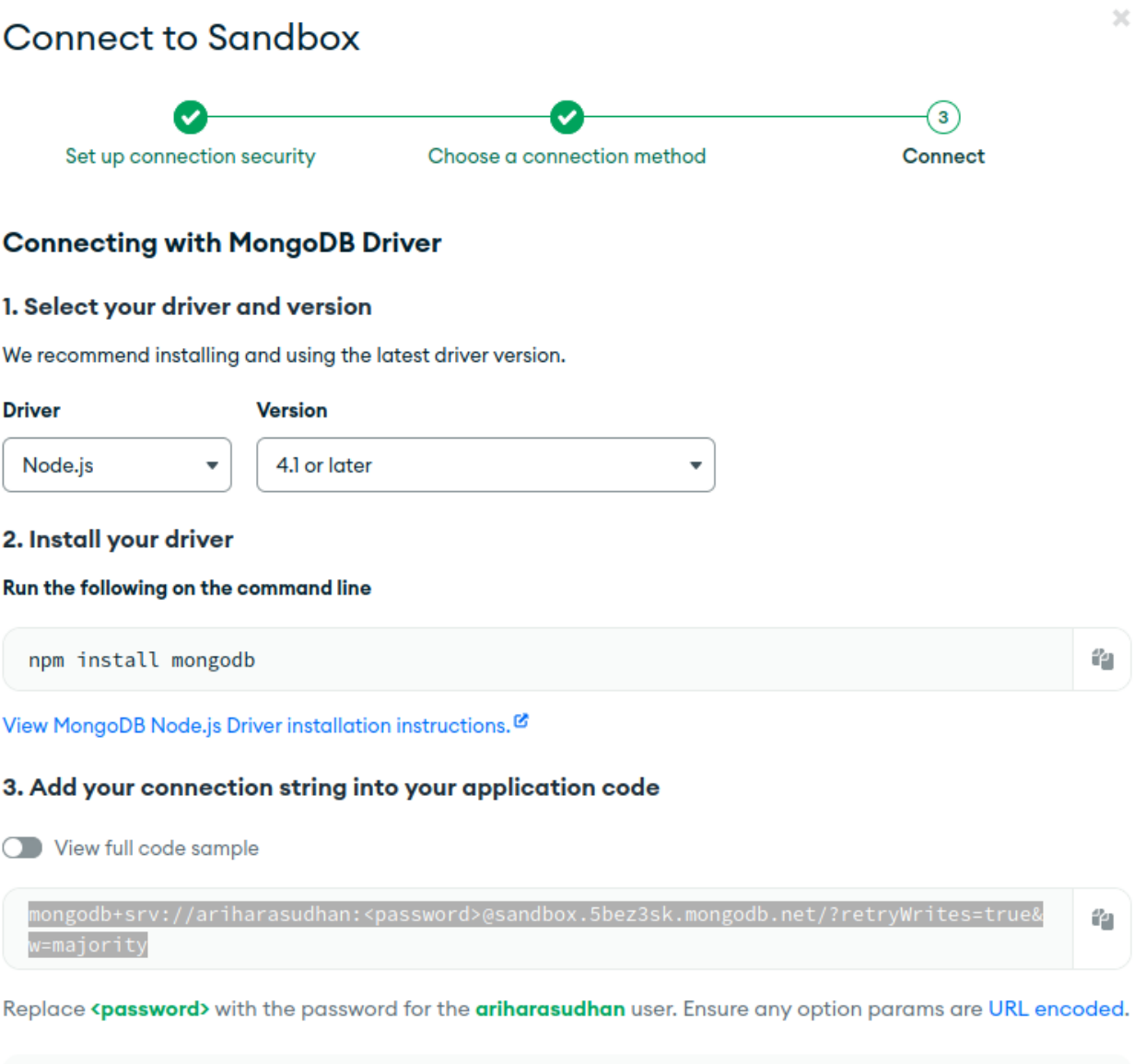



You have to give a username and a password for database creation. Make sure you remember those for connecting. Give a catchy-simple password.

Click Connect and click Drivers under Connect to your application.




You will be given your connection string. Copy that and replace the <password> with your password. That's it with MongoDB.

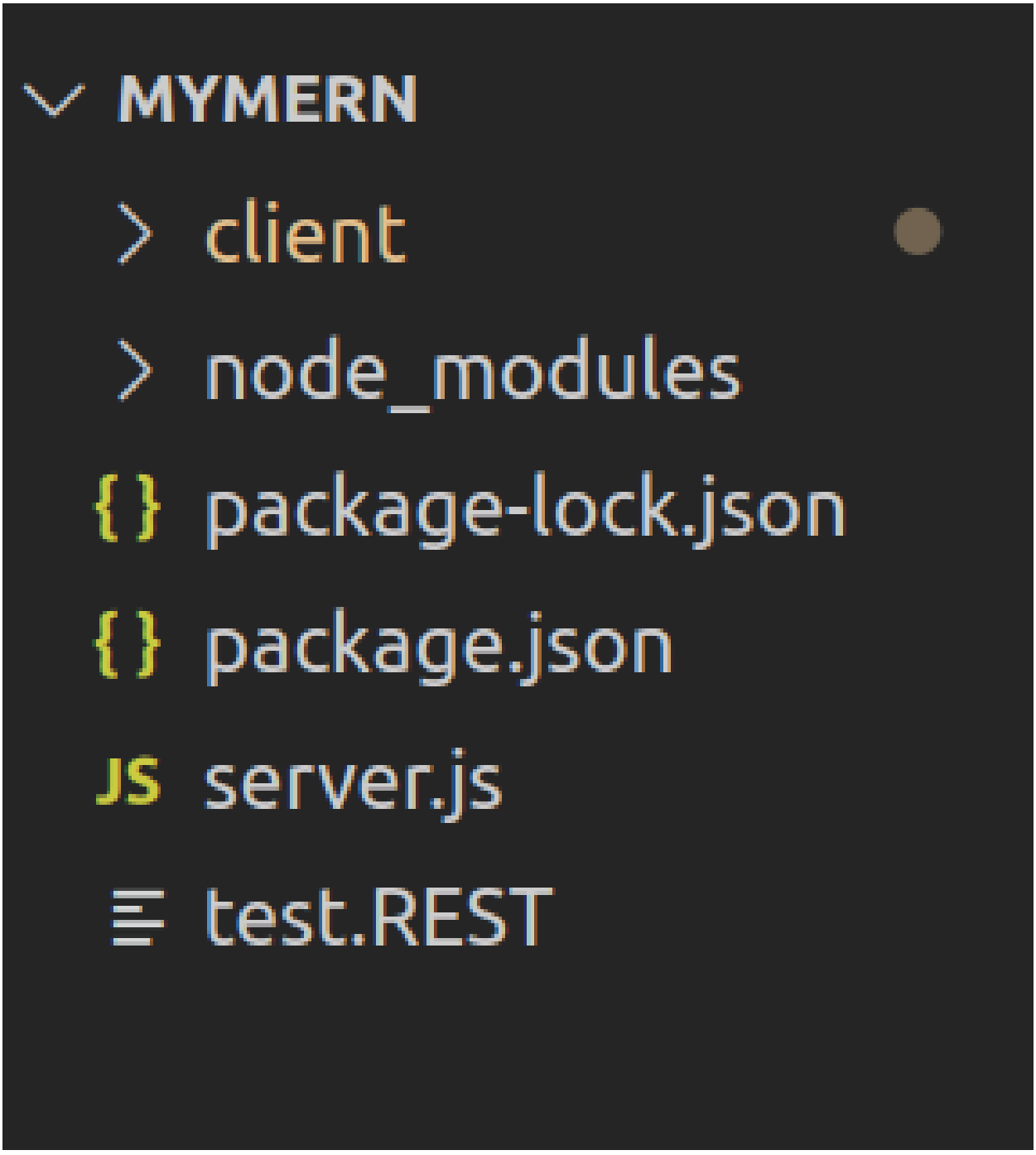



Now, we have some steps to follow. Let's simply play with the Full Stack Operations. We already have a server which is yet to be configured well. We need something called, Mongoose. Mongoose is widely used nowadays for MongoDB oriented operations.

# Simple MERN Stack App

Let's get familiar with how to do CRUD Operations in MERN Stack and do a simple MERN Stack Application.

Our full-stack directory will follow the following structure.



# THE BACKEND

(1) Let's create a folder mymern

```
> mkdir mymern
> cd mymern
```

(2) Let's initialize the project

```
> npm init -y
```

Now, you will be able to see package.json generated. It holds the information / meta-data of our server. Yes! We are currently creating a server file.

(3) Let's install necessary libraries

```
> npm install express body-parser cors mongoose
```

Now, you will be able to see package-lock.json generated. It holds the information / meta-data of our installed libraries.

(3) Let's install necessary libraries

```
> npm install express body-parser cors mongoose
```

Now, you will be able to see package-lock.json generated. It holds the information / meta-data of our installed libraries.

(4) Let's create a file called server.js. You can name it as per your wish. But, make sure to change in the main of package.json

```
"main": "server.js",
```

(5) In server.js, import the required libraries.

```
const express = require('express');
const mongoose = require('mongoose');
const bodyparser = require('body-parser');
const cors = require('cors');
```

(6) Create an Express App ( This is our server app ) and enable some middleware supports like json and cors. [ CORS : It is to whitelist to whomever the backend service should be provided. Let it blank for now (Anyone can access the backend) ]

```
const app = express();
app.use(bodyparser.json());
app.use(cors());
```

(7) Get the connection string from MongoDB and connect. Make sure to have a database with the name, “mymern” in MongoDB.

```
mongoose.connect("mongodb+srv://ariharasudhan:tiger@sandbox.5bez3sk.mongodb.net/?retryWrites=true&w=majority",
{ useNewUrlParser: true, useUnifiedTopology: true, dbName: 'mymern' })
.then(()=>{
    console.log("CONNECTED TO MONGODB")
})
.catch((err)=>{
    console.log(err)
});
```

(8) Now, I am about to insert a student record into the students collection of mymern database. So, make sure to create a collection with the name, students. A student record will be a name and age. So, let’s define a schema for it. Schema is like a structure of a record we insert into the database collection.

```
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number
});

const Student = mongoose.model('students',studentSchema);
```

The step we have followed after we create a schema is to tell, to which collection in the database the studentSchema is for.

(9) Now, It's time for routing. We will be do only CREATING (POST) and READING (GET) of students. You can try DELETING (DELETE) and EDITING (PUT). When we go to HOST/getStudents, the student records will be returned.

```
app.get('/getStudents',(req,res)=>{
    Student.find({})
    .then(result => {
      res.send(result)
    })
    .catch(err => {
        console.log(err)
    });

});

app.post('/createStudent',(req,res)=>{
    const newStudent = new Student({
        name: req.body.name,
        age: req.body.age
      });

      newStudent.save()
  .then(result => {
    res.send(result)
  })
  .catch(err => {
    console.log("ERROR");
  });
});
```

(10) The step left to complete the backend is to make the application listen to us.

```
const PORT  = 5000;
app.listen(PORT,()=>{
    console.log("LISTENING")
});
```

And now, NOTHING LEFT!

We can check out the backend with REST CLIENT Extension of Visual Studio as shown below.

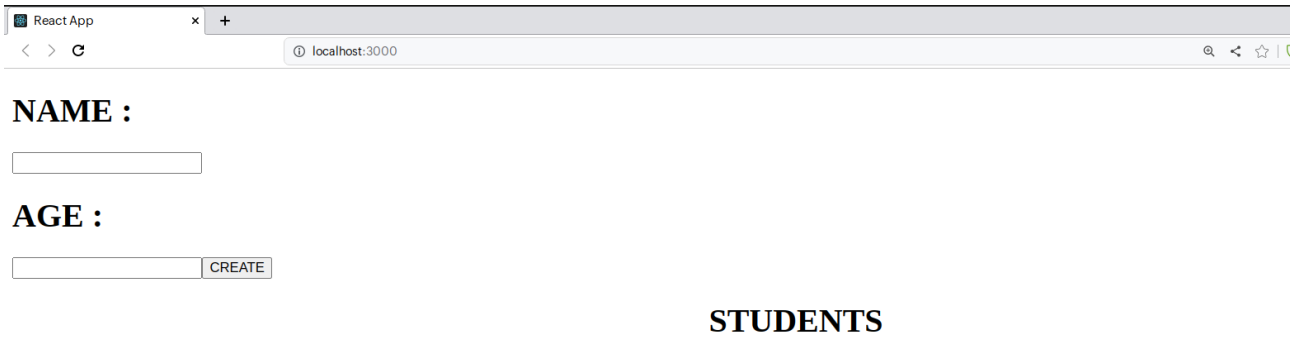
```
Send Request
POST http://localhost:5000/createStudent
Content-Type: application/json

{
    "name": "Karan",
    "age": 20
}

###
Send Request
GET http://localhost:5000/getStudents
```

# THE FRONTEND

Let's see some simple page creation to accept a student name and student age as shown below.




When we add a record using this React Page, a new record will be created there in MongoDB.

Create a React App with the name of client inside mymern directory using npx command. Install axios into the client folder using npm install axios. Delete unnecessary files. In your App.js, import the required.

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
```

Use useState Hook wisely. You can change it to your own style of programming.

```
function App() {
  const [name,setName] = useState('');
  const [age,setAge] = useState(-1);
  const [data,setData] = useState([]);
```

Here we are.. This is so important and cool. Use axios to connect with the backend for a POST request.

```
const addStudent = async () => {
  await axios.post('http://localhost:5000/createStudent', { name: name , age: age});
  setName('');
  setAge(-1);
  showStudents();
};
```

Read all the students from the server using GET request and store it in an array called data using setData.

```
useEffect(() => {
  showStudents();
}, []);

const showStudents = async ()=> {
  const resp = await axios.get('http://localhost:5000/getStudents');
  setData(resp.data);
}
```

Create a simple JSX component to show the essential stuffs.

```
  return (
    <div className="App">
      <h1>NAME : </h1>
      <input type='text' onChange={(e) => setName(e.target.value)}/>
      <h1>AGE : </h1>
      <input type='number' onChange={(e) => setAge(e.target.value)} />
      <button onClick={addStudent}>CREATE</button>
      <center>
      <h1>STUDENTS</h1>
      <ul>
        {data.map((item) => (
          <li>{item.name} is {item.age} years old</li>
        ))}
      </ul>
      </center>
    </div>
  );
}

export default App;
```

We have used the map to map each student to the specified JSX and that's all the MERN stack! You can furtherly grow with the knowledge of how to host it on render or any sort of platforms. On having time, have a nice visit to my blog made with MERN stack development.

I AM HERE : southernboy.onrender.com

AND ALSO : arihara-sudhan.github.io