Table of Contents

[**part 01 – Register Form** 2](#_Toc73974051)

[**part 02 – Preparing Form Values** 4](#_Toc73974052)

[**part 03 – Storing Form Values** 7](#_Toc73974053)

[**part 04 – Fixing the Menu** 10](#_Toc73974054)

[**part 05 – Logging In** 11](#_Toc73974055)

[**part 07 – Verify User** 13](#_Toc73974056)

[**part 08 – Updating Login Menu Item** 14](#_Toc73974057)

[**part 09 – Logging Out** 16](#_Toc73974058)

[**Appendix A – CORS Browser Plugin** 18](#_Toc73974059)

[**Appendix B – The this Operator in ReactJS** 21](#_Toc73974060)

# part 01 – Register Form

We would need a component in ReactJS to register a new customer.

1. Duplicate the **home** folder and change the name to **register**. Also change home.js to register.js and of course change the name of the function:

|  |
| --- |
|  |

1. Before we continue, lets add this new **register** component to the router system, so in App.js:

|  |
| --- |
| **import Home from './components/home/home';**  **import Customers from './components/customers/customers';**  **import Register from './components/register/register';**  **import { BrowserRouter, Route } from 'react-router-dom';**  **function App() {**  **return (**  **<div>**  **<BrowserRouter>**  **<Route exact path="/" component={ Home } />**  **<Route path="/customers" component={ Customers } />**  **<Route path="/register" component={ Register } />**  **</BrowserRouter>**  **</div>** |

4. In main.js within the register folder remove the code from home.js and add this form code:

|  |
| --- |
| **function Main(){**  **return(**  **<main>**  **<h2>Register for the Competition Here</h2>**  **<form>**  **<div>**  **<label>**  **Name:**  **<input type="text" name="username" />**  **</label>**  **</div>**  **<div>**  **<label>**  **Password:**  **<input type="text" name="password" />**  **</label>**  **</div>**  **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **</form>**  **</main>**  **);**  **}** |

5. Check the path and the form via the url:

|  |
| --- |
|  |

6. One last thing before we do the post, lets turn our function Main into a class Main. Although we can now accomplish the post request with a function, we will use a class in this boot camp:

|  |
| --- |
| **import React, { Component } from "react";**  **class Main extends Component{**  **render();**  **return(**  **<main>**  **<h2>Register for the Competition Here</h2>**  **<form>** |

The class must extend **Component**, which will probably be inserted for you by VS Code, if not you must import it via destructuring from the react package. Also a class must *render* code not just *return* something.

7. now cut the entire return(…) code and paste it between the { } if the render() method:

|  |
| --- |
| **class Main extends Component{**  **render(){**  **return(**  **<main>**  **<h2>Register for the Competition Here</h2>**  **<form>**  **<div>**  **<label>**  **Name:**  **<input type="text" name="username" />**  **</label>**  **</div>**  **<div>**  **<label>**  **Password:**  **<input type="text" name="password" />**  **</label>**  **</div>**  **<div>**  **<input type="submit" value="Submit" />**  **</div>**  **</form>**  **</main>**  **);**  **};**  **}** |

This is the entire class so far inside of the main.js file in the register folder

# part 02 – Preparing Form Values

1. In order to use ReactJS **state**, we need a constructor in this class that references the **props** property via a call to **super().**

|  |
| --- |
| **class Main extends Component{**  **constructor(props) {**  **super(props);**  **}**  **render(){**  **return** |

Note you may have to include this line in the class to avoid certain linting errors:  **// eslint-disable-next-line no-useless-constructor**

1. At the same time we can define a few variables to hold state as we conduct busiiness

|  |
| --- |
| **constructor(props) {**  **super(props);**  **this.state = {**  **username:"",**  **password:""**  **}**  **}**  **render(){**  **return(** |

1. Lets add some functionality to the **onSubmit** event. We will define a function in this class called **handleSubmit()**

|  |
| --- |
| **<main>**  **<h2>Login</h2>**  **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label> })**  **}**  **//** |

1. Add a function inside of the class which will invoke the **fetch()** method of JS. You can do this above the **render()** function. Also add in the **preventDefault()** call to avoid the form be submitted without values.

|  |
| --- |
| **}**  **}**  **handleSubmit(event) {  event.preventDefault();**  **fetch()**  **}**  **render() {**  **return (** |

1. With that in place, we need to register this new function with the *this* reference, in our constructor. If we did not do this, the *this* keyword would refer to the Window object and not our component. The *this* keyword in JS is relevant to the context in which it is used. We do this in the constructor.

|  |
| --- |
| **constructor(props) {**  **super(props);**  **this.handleSubmit = this.handleSubmit.bind(this);**  **this.state = {**  **username:"",**  **password:""**  **}** |

Note:there are other methods of handling this binding issue

1. The fetch() method can also make POST requests, however we have to add a second configuration object as a parameter. This configuration object will contain everything we need in order for fetch() to complete the request properly:

|  |
| --- |
| **}**  **handleSubmit(event) {**  **event.preventDefault();**  **fetch('http://localhost:3030/customers',{})**  **}**  **render(){** |

1. Also add in the **preventDefault()** call to avoid the form be submitted without values.
2. We pass into the configuration object the key/value pairs that are important right now:

|  |
| --- |
| **handleSubmit(event) {**  **event.preventDefault();**  **fetch(**  **'http://localhost:3030/customers',**  **{**  **method: 'POST',**  **headers: { },**  **body: JSON.stringify({ })**  **}**  **)**  **}** |

1. We now complete the headers section first, so we want to pass json data to the server:

|  |
| --- |
| **fetch(**  **'http://localhost:3030/customers',**  **{**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({ })**  **}** |

The Accept request HTTP header is telling the server which MIME types the browser is able to handle. Content type refers to the type of file being transferred via HTTP and classified by IANA.

1. We now complete the body section:

|  |
| --- |
| **fetch(**  **'http://localhost:3030/customers',**  **{**  **method: 'POST',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **},**  **body: JSON.stringify({**  **username: this.state.username,**  **password:this.state.password**  **})**  **}**  **)** |

1. (Optional) if you wanted to see the data being posted, do this before you code the fetch() method:

|  |
| --- |
| **handleSubmit(event) {**  **event.preventDefault();**  **console.log(**  **JSON.stringify({**  **username: this.state.username,**  **password:this.state.password**  **})**  **);**  **}** |

# part 03 – Storing Form Values

1. Before we post anything, we need the values that the user types into the input boxes of the HTML form, so we do this with the *onChange* event of the form control. When that change happens we provide a function to handle that change, which really means storing the vaue inside the **state object**. So first lets write a function function to handle these values, you can insert it anywhere in the class.

|  |
| --- |
| **handleFieldChange(event) {**  **this.setState({**  **[event.target.name]:event.target.value**  **})**  **}** |

1. We will pass this function to the **onChange** event of the field. We capture the value using the event object. The word *event* can be anything, it can be just ‘e’. However with that object we can access both the name of the field and its value. Once we get the values and the field names, we invoke **setState()** to register these objects with our state object.
2. Just like with **handleSubmit**, we also have to register **handleFieldChange** in our constructor

|  |
| --- |
| **class LoginMain extends Component{**  **constructor(props) {**  **super(props);**  **this.handleSubmit = this.handleSubmit.bind(this);**  **this.handleFieldChange = this.handleFieldChange.bind(this);**  **this.state = {**  **empName:"",**  **empWeight:"",** |

1. Now we can implement this function in the HTML

|  |
| --- |
| **<form onSubmit={this.handleSubmit}>**  **<div>**  **<label>**  **Name:**  **<input type="text" name="username" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>**  **<label>**  **Password:**  **<input type="text" name="password" onChange={this.handleFieldChange}/>**  **</label>**  **</div>**  **<div>** |

1. Lets try to register a new customer:

|  |
| --- |
|  |

Notice we get a CORS error

1. Although the error is point to CORS it is actually that our fake back end server is unable to handle the request. The data is not structured right, we need to add some kind if unique id to the data set:

|  |
| --- |
| **{**  **"username": "Axle",**  **"password": "1234",**  **"id": "1"**  **},**  **{**  **"username": "Jane",**  **"password": "1234",**  **"id": "2"**  **},**  **{**  **"username": "Mary",**  **"password": "1234",**  **"id": "3"**  **}** |

Just add an id key value pair, make sure the value part is unique

1. Clear the browser error and try adding a new customer again, check the json-server url to make sure the customer was added:

|  |
| --- |
|  |

Notice that Bob was given a unique ID by the server, not our code

1. Before ending this section continue the fetch method to report a successful post or if an error occurred:

|  |
| --- |
| **username: this.state.username,**  **password:this.state.password**  **})**  **}**  **).then().catch();**  **}**  **render(){**  **return(** |

1. We can now pass in functions into both of these chained methods to report something:

|  |
| --- |
| **}**  **)**  **.then(response => console.log("Post successful " + response))**  **.catch(err => console.log("Error occured " + err.message));**  **}**  **render(){** |

# part 04 – Fixing the Menu

Now that we have three components, lets change the menu to reflect these, so home, register and customers

1. In header.js, now in the wrapper folder, import the **Link** component from react-router-dom

|  |
| --- |
| **import React from "react";**  **import { Link } from 'react-router-dom';**  **import logo from '../../chart.gif';**  **function Header(){** |

1. Now all we need to do is use the **Link to** syntax to point our various menu items to the correct ‘view’

|  |
| --- |
| **<nav>**  **<ul>**  **<li><Link to="/">home</Link></li>**  **<li><Link to="/register">register</Link></li>**  **<li><Link to="/customers">customers</Link></li>**  **</ul>**  **</nav>** |

1. Although the default path works for home, we should hard code an actual home path. In App.js add this line:

|  |
| --- |
| **<div>**  **<BrowserRouter>**  **<Route exact path="/" component={ Home } />**  **<Route path="/home" component={ Home } />**  **<Route path="/customers" component={ Customers } />**  **<Route path="/register" component={ Register } />**  **</BrowserRouter>**  **</div>** |

1. While we are here lets add one more link to be able to login our customers

|  |
| --- |
| **import Register from './components/register/register';**  **import Login from './components/login/login';**  **import { BrowserRouter, Route } from 'react-router-dom';**  **function App() {**  **return (**  **<div>**  **<BrowserRouter>**  **<Route exact path="/" component={ Home } />**  **<Route path="/home" component={ Home } />**  **<Route path="/customers" component={ Customers } />**  **<Route path="/register" component={ Register } />**  **<Route path="/login" component={ Login } />**  **</BrowserRouter>** |

Of course this component does not yet exist so this change will throw an error until we build that component. For now you can comment these lines until we build the component.

# part 05 – Logging In

We can now create a new login component to log the users in. Since the login function will use the same form as the register, just duplicate the register component and do some cosmetic changes.

1. If you duplicated the register component then you must change the register.js to login.js inside of the login folder, also change the function name and export:

|  |
| --- |
|  |

1. In App.js re-activate the login component if you had it there but commented it out:

|  |
| --- |
| **import Register from './components/register/register';**  **import Login from './components/login/login';**  **import { BrowserRouter, Route } from 'react-router-dom';**  **function App() {**  **return (**  **<div>**  **<BrowserRouter>**  **<Route exact path="/" component={ Home } />**  **<Route path="/home" component={ Home } />**  **<Route path="/customers" component={ Customers } />**  **<Route path="/register" component={ Register } />**  **<Route path="/login" component={ Login } />**  **</BrowserRouter>** |

1. In the header.js file iside of the wrapper folder, add the new login link:

|  |
| --- |
| **<header>**  **<img src={logo} id="logo" alt="logo"/>**  **<h1><a href="index.html">Skillsoft Weight Tracker</a></h1>**  **<nav>**  **<ul>**  **<li><Link to="/">home</Link></li>**  **<li><Link to="/register">register</Link></li>**  **<li><Link to="/customers">customers</Link></li>**  **<li><Link to="/login">login</Link></li>**  **</ul>** |

At this point, both register and login should show the same content.

1. Change the <h2> tag in main.js inside of login folder to reflect that the form is being used as a login form and NOT to register the customer

|  |
| --- |
| **render(){**  **return(**  **<main>**  **<h2>Login to post your weight</h2>**  **<form onSubmit={this.handleSubmit}>**  **<div>** |

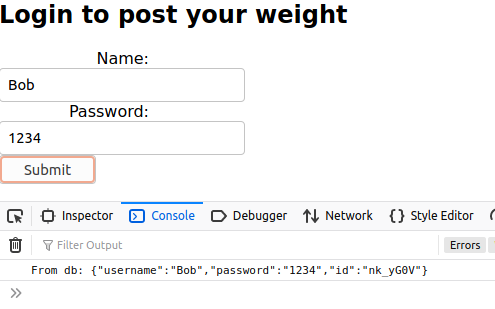
1. We need to change the **handleSubmit()** method of our login main.js file. That method should pass the username and retrieve that user if the user exists. First change the method to a **GET**, remove the **body object** and wrap the **username** as shown:

|  |
| --- |
| **handleSubmit(event) {**  **event.preventDefault();**  **fetch(**  **`http://localhost:3030/customers?username=${encodeURIComponent(this.state.username)}`?,**  **{**  **method: 'GET',**  **headers: {**  **'Accept': 'application/json',**  **'Content-Type': 'application/json'**  **}**  **}**  **)**  **.then(response => console.log("Post successful " + response))**  **.catch(err => console.log("Error occured " + err.message));**  **} )** |

Note: the quotation marks are the backtick character NOT single or double quotes

1. Before moving forward, just check that you are getting the user, we would need to first return the object we get via the fetch() method by extracting its json data, then logging the database object NOT the object sent via the form itself:

|  |
| --- |
| **.then(response => response.json())**  **.then(data => console.log("From db: " + JSON.stringify(data[0])))**  **.catch(err => console.log("Error occured " + err.message));** |



# part 07 – Verify User

1. Now that we can access the user stored in our database we can test the users stored password against the one supplied via the form. Replace the console.log() statement with a function that handles the verification:

|  |
| --- |
| **)**  **.then(response => response.json())**  **.then(data => {**  **if(data[0] === this.state.password){**  **//we have a match**  **}**  **})**  **.catch(err => console.log("Error occured " + err.message));**  **}** |

If these two values match, we have a valid user

1. (Optional) You can also code for an invalid login

|  |
| --- |
| **.then(data => {**  **if(data[0].password === this.state.password){**  **//we have a match**  **} else {**  **console.log("Invalid login");**  **}**  **})**  **.catch(err => console.log("Error occured " + err.message));** |

1. If we have a valid customer, then one option is to store the user in local storage for future reference

|  |
| --- |
| **.then(data => {**  **if(data[0].password === this.state.password){**  **localStorage.setItem('validuser', this.state.username);**  **} else {**  **console.log("Invalid login");**  **}** |

1. At the same time we can send them back to the home page or any other page your business logic dictates:

|  |
| --- |
| **.then(data => {**  **if(data[0].password === this.state.password){**  **localStorage.setItem('validuser', this.state.username);**  **this.props.history.push("/");**  **} else {**  **console.log("Invalid login");**  **}**  **})** |

1. The above code will not work as is, we would need to import **withRouter** and also adjust the **export default** line:

|  |
| --- |
| **import React, { Component } from "react";**  **import { withRouter } from "react-router";**  **class Main extends Component{**  **// eslint-disable-next-line no-useless-constructor**    **constructor(props) { … (rest of the code)  </main>**  **);**  **};**  **}**  **//**  **export default withRouter(Main);** |

# part 08 – Updating Login Menu Item

1. One way to remove the login menu item is to pass a message from the login component back to the App.js file. This involves a lot of work so we will simply check local storage and take it from there. First in header.js, change the function to a class and add a constructor:

|  |
| --- |
| **import { Link } from 'react-router-dom';**  **class Header extends Component{**  **//**  **constructor(props) {**  **super(props);**  **}**  **//**  **render(){**  **return(** |

Remember to de-structure Component when you import React (VSCode does this automatically)

1. Add a state property to check for validuser:

|  |
| --- |
| **constructor(props) {**  **super(props);**  **this.state = {validUser: localStorage.getItem('validuser')};**  **}** |

Note, I am using validUser to distinguish from validuser

1. Simply change the nav element to check for validuser

|  |
| --- |
| **<nav>**  **<ul>**  **<li><Link to="/">home</Link></li>**  **<li><Link to="/register">register</Link></li>**  **<li><Link to="/customers">customers</Link></li>**  **{ !this.state.validUser**  **? <li><Link to="/login">login</Link></li>**  **: null**  **}**  **</ul>**  **</nav>** |

The above code works but you may want to pass a logout link instead of just null

1. Adding a logout link

|  |
| --- |
| **<li><Link to="/customers">customers</Link></li>**  **{ !this.state.validUser**  **? <li><Link to="/login">login</Link></li>**  **: <li><Link to="/logout">logout</Link></li>**  **}**  **</ul>**  **render(){** |

1. Insert the user name to make it more personal

|  |
| --- |
| **{ !this.state.validUser**  **? <li><Link to="/login">login</Link></li>**  **: <li><Link to="/logout">logout {this.state.validUser}</Link></li>**  **}** |

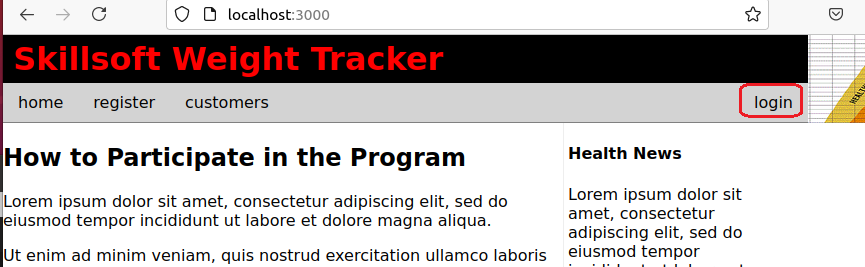
1. (Optional) You can add a style to move the login/logout links to the right of the menu

|  |
| --- |
| **.moveRight{**  **float: right;**  **}** |

Add this class in styles.css

1. (Optional) back in header.js in the render/return section, add the class there:

|  |
| --- |
| **{ !this.state.validUser**  **? <li className="moveRight"><Link to="/login">login</Link></li>**  **: <li className="moveRight"><Link to="/logout">logout {this.state.validUser}</Link></li>**  **}** |



# part 09 – Logging Out

1. Since most of the logging activities happens in the Header component, we only need to update this file. First add an **onClick()** event to the logout element, pointing it to a function we will write shortly

|  |
| --- |
| **{ !this.state.validUser  ? <li className="moveRight"><Link to="/login">login</Link></li>**  **: <li className="moveRight" onClick={this.logOut}><Link to="/logout">logout this.state.validUser}</Link></li>**  **}** |

1. Begin the **logout()** function and of course bind it in the constructor

|  |
| --- |
| **iconstructor(props) {**  **super(props);**  **this.state = {validUser: localStorage.getItem('validuser')};**  **this.logOut = this.logOut.bind(this);**  **}**  **//**  **logOut(){**  **}** |

1. Now all you have to do is delete the validuser key and value from local storage and re-direct the browser to a different view

|  |
| --- |
| **logOut(){**  **localStorage.removeItem('validuser');**  **this.props.history.push('/home');**  **}** |

1. For the re-direction, import the withRouter module from react-router-dom

|  |
| --- |
| **import { Link, withRouter } from 'react-router-dom';**  **class Header extends Component{** |

1. Then when you export this class, export it withRouter

|  |
| --- |
| **}**  **//**  **export default withRouter(Header);** |

1. There is still a problem with this setup, if you leave the url as **/home** then logout you will clear the local storage but the logout menu will remain:

|  |
| --- |
|  |

1. We already know that state properties cannot be directly updated so we are supposed to use **setState()**

|  |
| --- |
| **ilogOut(){**  **localStorage.removeItem('validuser');**  **this.setState = ({validUser: null});**  **this.props.history.push('/home');**  **}** |

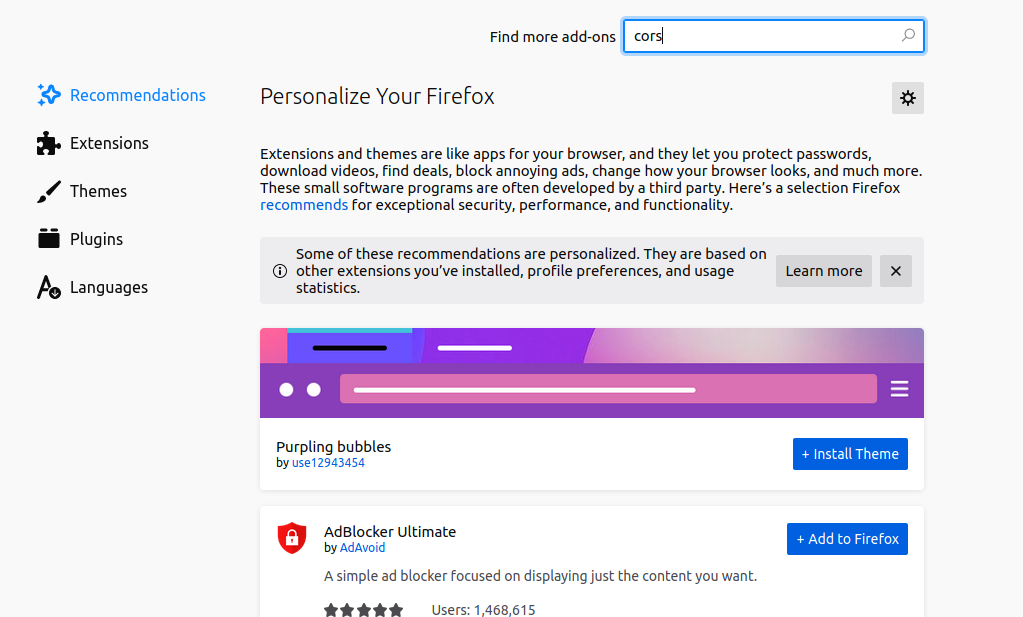
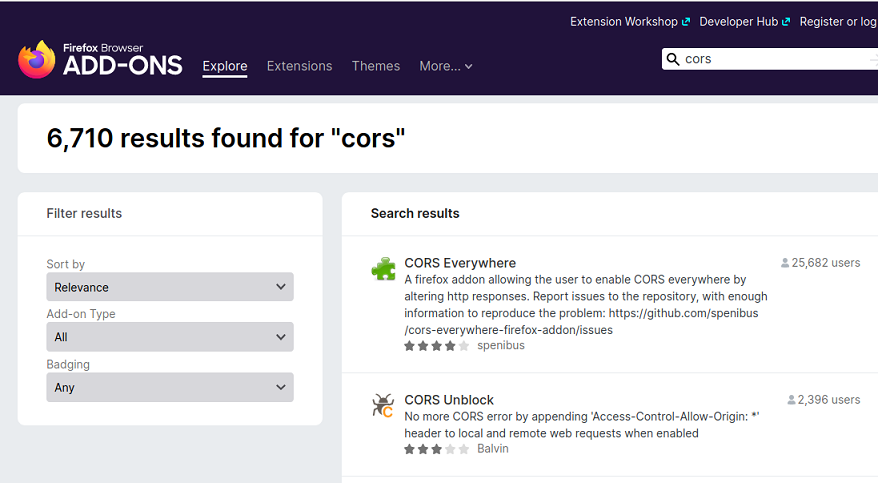
However this also will not work

1. **setState()** will not work in ths situation as it does not know the previous state, so we need to pass the current state like this:

|  |
| --- |
| **logOut(){**  **localStorage.removeItem('validuser');**  **this.setState((state) => ({**  **validUser: null**  **}));**  **this.props.history.push('/home');**  **}** |

# Appendix A – CORS Browser Plugin

Sometimes during development, it is necessary to get around CORS errors. One way is to use code, the other is to simply install a plugin for your browser. Below I describe how to add a plugin for Firefox.

1. Lets add a browser plugin to overcome any CORES issue. If you are using Firefox, click on the hamburger icon on the top right corner, then go to **Add-ons and Themes**, from there search for CORS:  
     
     
   
2. Click on CORS Everywhere to install it   
   
3. Click on the big blue Add to Firefox button to add the plugin, accept all the requirements

|  |
| --- |
|  |

1. Turn on the plugin by clicking on it in the browser, it will turn from pink to green:

|  |
| --- |
|  |

# Appendix B – The this Operator in ReactJS

For a detailed explanation of how the this operator is used in ReactJs, refer to the linke below:  
<https://reactjs.org/docs/handling-events.html>

As an example, the handleFieldChange function can be written differently and thus avoid the binding of the this keyword, however this option is NOT suggested.

|  |
| --- |
| **constructor(props) {**  **super(props);**  **this.handleSubmit = this.handleSubmit.bind(this);**  **//this.handleFieldChange = this.handleFieldChange.bind(this);**  **this.state = {**  **username:"",**  **password:""**  **}**  **}**  **handleFieldChange = (event) => {**  **this.setState({**  **[event.target.name]:event.target.value**  **})**  **}**  **handleSubmit(event) {** |