**Working with React Form and Handling Event**

[In the previous part](https://ibaslogic.com/react-tutorial-for-beginners/), we covered some of the React basics; set up React working environment and started writing our Todos application.

**This React tutorial is part 2 of 11 in the React for beginners series.**

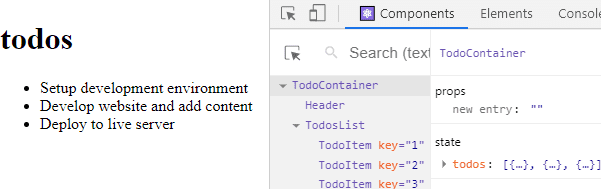
* Part 1 – [React Tutorial: The Beginner's Guide to Learning React in 2020](https://ibaslogic.com/react-tutorial-for-beginners/)
* Part 3 – [How to implement CSS in Reactjs App](https://ibaslogic.com/css-in-reactjs-app/)
* Part 4 – [How to Edit Todos Items](https://ibaslogic.com/how-to-edit-todos-items-in-react/)
* Part 5 – [Persisting React State in Local Storage](https://ibaslogic.com/persisting-react-state-in-local-storage/)
* Part 6 – [Getting Started With React Lifecycle Methods](https://ibaslogic.com/react-lifecycle-methods/)
* Part 7 – [Getting Started With React Hooks](https://ibaslogic.com/react-hooks-tutorial/)
* Part 8 – [How to use SVG Icons in React](https://ibaslogic.com/how-to-use-svg-icons-in-react-project/)
* Part 9 – [Routing With React Router](https://ibaslogic.com/routing-with-react-router/)
* Part 10 – [How to add Hamburger Menu in React](https://ibaslogic.com/how-to-add-hamburger-menu-in-react/)
* Part 11 – [Deploying React App to GitHub Pages](https://ibaslogic.com/deploying-react-app-to-github-pages/)

Now, in this part, we will take a look at integrating form in our React app. Also, you will get to know how to raise and handle events in React.

Though, we’ve covered [how to handle the form inputs fields](https://ibaslogic.com/simple-guide-to-react-form/) using the modern React Hooks. There, you will learn how the common input types such as checkbox, text, textarea, select input, radio and range work in React.

Until you’ve learned what these Hooks are in this series, you can bookmark the page for later read. But, if you have a basic knowledge of the Hooks, you can read to have prior knowledge of how the form works in React.

Let’s move on. Here is the current status of our app.



And Now, let’s start by adding the checkboxes.

**Adding checkboxes to the Todo items**

You should be familiar with handling this type of input field in the regular HTML form. In React, form handling works a little bit different.

So let's see how to.

Open the TodoItem.js file and add the checkbox input just before the title in the li element.

return (

<li>

<input type="checkbox" /> {this.props.todo.title} </li>

)

Save the file and see the checkboxes in the frontend.

By default, the input type (i.e checkboxes) are being handled by the DOM – i.e they have the default HTML behaviour. That is why you can toggle the boxes.

This type of input is called **uncontrolled input**. But this is not the case with React. The input fields are meant to be **controlled**. This takes us to another important subtopic.

**Controlled Component**

To make the input field controllable, the input data (in this case, toggling of the checkbox) has to be handled by the component state and not the browser DOM. With this, the state will serve as a **Single Source of Truth**.

Meaning, the input checkbox would no longer listens to its internal state (i.e the browser DOM) but the state in your app. This is necessary because the component state will not change unless you change it.

Let’s see how it works. If you take a look at the state object in the parent component, we have a Boolean value (true or false) assigned to every completed key in the todos data. We can tap into that to toggle the checkboxes.

So go inside the TodoItem.js file and add a checked prop to the input checkbox. Then assign the Boolean value from the state object.

Remember, we can access these values the same way we accessed the title. The checkbox input should look like this:

<input type="checkbox" checked={this.props.todo.completed} />

Save the file.

At this point, you have succeeded in making the input checkbox a controlled input because it now listens only to the state in your application.

Now if you try to toggle any of the checkboxes, nothing will happen. This is because each of the checked attributes is assigned a value equal to the current value of the state.

Remember, only the first task is assigned to be completed. We need a way to change the state whenever users click on the checkboxes. React already gives us a hint through the Console tab of the browser DevTools.

If you open it, you'll see a warning displayed as a result of the added checked attribute. React is telling us to add an onChange handler to keep track of any changes in the field. Else, it wouldn’t know if the input field is checked or not.

So let’s update the input tag in the TodoItem.js file to include the handler.

<input

type="checkbox"

checked={this.props.todo.completed}

onChange={() => console.log("clicked")}/>

Save the file and look inside the Console tab once again, you'll see that the warning is gone.

For the meantime, we are assigning to the handler, a callback function that will log a "clicked" text in the console whenever the checkbox is clicked.

Open the console and try to click on the checkboxes. Now, instead of logging text in the console, we want to toggle the checkboxes anytime they are being clicked. To do this, we need to understand how to raise and handle events.

**Raising and Handling Events**

In our app, the parent component, TodoContainer is the one that holds the state data. This component, therefore, is the ONLY one that can change it. Meaning the TodoItem component, which is the one handling the checkboxes, cannot change the state data in the parent component, TodoContainer.

We need to find a way to access the state data from the TodoItem and toggle the completed value to true or false in the TodoContainer component.

To do this, we will need to raise an event from the TodoItem up a level to TodosList, and then into TodoContainer component. In other words, we need to **climb a ladder**.



The TodoItem component will raise the event while the parent component, TodoContainer will handle the event. And the way we do that is through props.

This is kind of tricky but trust me it's very simple. You can either go from the child to parent component or the other way round. I prefer the latter.

So let’s do it. We will first enable communication between these components. Starting from the parent component, TodoContainer, add a handler method, handleChange just above the render() method.

handleChange = () => {

console.log("clicked");

};

You can name this method anything you like. Let’s see how we can communicate with this method from the child component.

Start by passing this method to the TodosList component through the props. So update <TodosList /> so you have:

<TodosList todos={this.state.todos} handleChangeProps={this.handleChange} />

**Note:** We are using this.handleChange to reference the handleChange() method because it is part of the class.

Now, you have the handleChange() method assigned to the handleChangeProps. Its data can be accessed through props in the TodosList component.

From there, we can pass it to the TodoItem component. Let’s update the <TodoItem /> instance in the TodosList.js file so you have:

<TodoItem

key={todo.id}

todo={todo}

handleChangeProps={this.props.handleChangeProps}

/>

At this point, the handleChange() data can be accessed from the TodoItem component. So update the onChange handler in the TodoItem component so you have:

onChange={() => this.props.handleChangeProps()}

This time, make sure you have parenthesis, () attached to the handleChangeProps. Save all your files. Now, if you click any of the checkboxes, the onChange event will trigger and will call the handleChange() method in the parent component, TodoContainer.

For now, we are only logging a text in the console.

Let’s go a step further. We need to identify which one of the checkboxes is clicked. To do this, we need to pass along their respective ids through the callback function.

Update the onChange handler in the TodoItem component to include the id.

onChange={() => this.props.handleChangeProps(this.props.todo.id)}

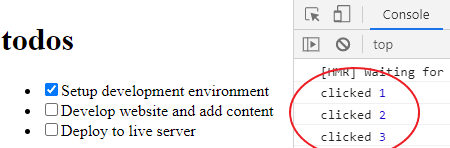
Remember, just like the title and the completed value, we also have access to the id in this component. Save the file. Then go inside the TodoContainer component and update the handleChange method.

handleChange = (id) => {

console.log("clicked", id);

};

Note how we are receiving and logging the id. Save the file. Open the console and click on the checkboxes. This time, you will see their respective ids.



Good. We are heading somewhere.

**Updating the state using the setState() method**

Our aim here is to change the state of the checkbox whenever it is clicked. In React, we do not modify the state directly. Instead, we update through a method we inherited by extending React.Component.

This method is called setState(). It tells React that we are updating the state. React would then schedule updates to the component local state and figures out what part of the state has changed. After that, it updates ONLY that part in the real DOM.

Let’s apply the method.

At the moment, if we click any of the checkboxes, we are receiving the clicked id in the handleChange. Now, for every toggle, the TodoContainer component will schedule a UI update by calling setState() with an object containing the current todos. Mind you, we will get the current todos by looping through the todos data and check if any of the items id matches the checked id. If it does, then we flip the completed value from true to false and vice versa.

Here is the code interpretation:

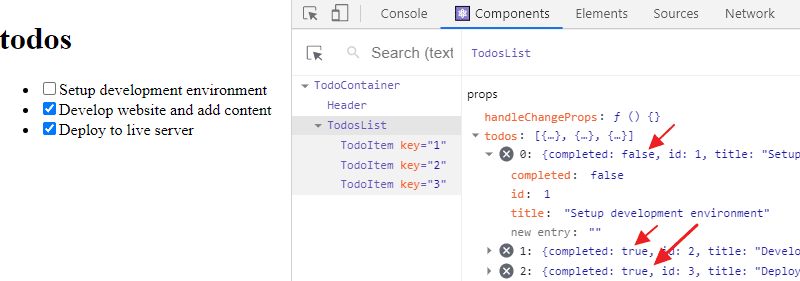
handleChange = id => {

this.setState({ todos: this.state.todos.map(todo => { if (todo.id === id) { todo.completed = !todo.completed; } return todo; }) });};

Make sure you update the handleChange method in the TodoContainer component to the code above.

Now, React knows through the setState() call that the state has changed. Then it calls the render() method once again to re-render and update the screen accordingly.

Save the file and check your application. You should be able to toggle the checkboxes.



You can also see the current state of your app in the React Tools.

**Using the setState updater**

You already know that the setState() method in a class component is used to update the state. However, the usage in the handleChange() can be optimized further. As we have it in our app, we are toggling a checkbox. And this can either be true or false.

In this case, the new state is computed based on the previous value. You can also find another scenario whenever you are updating a counter value or toggling a button.

So when this case arise, you can pass an updater function to the setState. The function will receive the previous version of the state, and return an updated value.

this.setState(prevState => ({

//update state

}))

The reason for this is that React does not guarantee that this.state written within the setState() is up-to-date. It may batch or defer the update until later.

So React advises not to rely on the value to calculate the next state as it may result in unintended output. Now, if you update the handeChange(), you should have:

this.setState(prevState => ({ todos: prevState.todos.map(todo => { if (todo.id === id) {

todo.completed = !todo.completed

}

return todo

}),

}))

In the code, we are now using the snapshot (prevState) within the setState() to get the previous state instead of using this.state.

Save your file and test your work.

There is a problem. We were not able to toggle the checkboxes anymore. If you temporarily remove the <React.StrictMode> in the src/index.js file and save it, the checkboxes start working again.

Since we want the strict mode enabled, please add it back.

In the TodoContainer.js file, we need to optimize the handleChange() method further.

At the moment, we are updating the checked value in the state directly through todo.complete = !todo.completed. The strict-mode doesn’t like that.

Instead let’s update the code so you have:

this.setState(prevState => ({

todos: prevState.todos.map(todo => {

if (todo.id === id) {

return {

...todo,

completed: !todo.completed,

}

}

return todo

}),

}))

Now, for every todos item and after the condition is satisfied, we are grabbing the item; spread its properties using the ES6 spread operator and then toggle only the completed value.

Save your file and test your work. You should be able to toggle the checkboxes again.

***A quick one:***

Note how we are wrapping the object in the setState callback with a parenthesis, (). An alternative is to use the return statement to explicitly return the object like so:

this.setState(prevState => {

return {

todos: prevState.todos.map(todo => {

if (todo.id === id) {

return {

...todo,

completed: !todo.completed,

}

}

return todo

}),

}

})

Both methods work fine.

**Deleting items from the todos**

This will be similar to how we handled the input checkboxes. Here also, the todos items live in the TodoContainer, while the delete button will live in the TodoItem component.

That means we will be raising an event from the TodoItem component and move up levels until we get to the TodoContainer component where the event will be handled.

Let’s get down. Start by adding a delete button in the TodoItem component. So, add this button element below the input tag (but before the title):

<button>Delete</button>

For the meantime, we are writing the "Delete" text instead of a proper delete icon. Later in the series, you’ll learn how to add SVG icons to your React application.

As usual, we will enable communication between these components to raise an event. So go inside the TodoContainer component and add a delTodo method above the render().

delTodo = id => {

console.log("deleted", id);

};

Still in the component, update the <TodosList /> to include:

deleteTodoProps={this.delTodo}

So you have:

<TodosList

todos={this.state.todos}

handleChangeProps={this.handleChange}

deleteTodoProps={this.delTodo}/>

Save the file and move a level down inside the TodosList component and update the <TodoItem /> so you have:

<TodoItem

key={todo.id}

todo={todo}

handleChangeProps={this.props.handleChangeProps}

deleteTodoProps={this.props.deleteTodoProps}/>

Finally, back in the TodoItem component. Update the button element to include an onClick event handler that will trigger the delTodo method in the parent component. You should have this:

<button onClick={() => this.props.deleteTodoProps(this.props.todo.id)}>

Delete

</button>

Save all your files and check the frontend. Open the Console tab and try to delete any of the todos items. At the moment, we are logging "deleted" text alongside the id of the deleted items.

Up to this point, we are repeating what we did for the checkbox. If it's not clear, revisit the earlier explanation.

Next, we will manipulate the state and remove any of the deleted items from the list. The way we do that is by using the filter() method. This method is also a higher-order function just like the map() method. It returns a new array by applying a condition on every array element.

In this case, we only want to return the todos items that do not match the id that will be passed in – i.e the clicked id. Any id that matches will be deleted. Now update the delTodo method so you have:

delTodo = id => {

this.setState({ todos: [ ...this.state.todos.filter(todo => { return todo.id !== id; }) ] });};

With the filter() method, we are saying that for each of the todos data that we are looping through, we want to retain the once whose id is not equal to the id passed in.

Please note the spread operator (…) in the code. It allows us to grab the current todos item(s) at every point. As this is necessary for the code to work.

Save the file and test your application. You should be able to delete any item(s) on the list.

Moving on.

**Adding a text input field and a submit button**

In React, all the different types of input fields follow the same approach. We’ve seen how the checkbox type of input field works. Using the same pattern, we will add the text input field that accepts the user's input.

Let’s start by adding the following code inside the empty InputTodo.js file:

import React, { Component } from "react"

class InputTodo extends Component {

render() {

return (

<form>

<input type="text" placeholder="Add Todo..." />

<button>Submit</button>

</form>

)

}

}

export default InputTodo

Note also that the "Submit" text will be replaced later by a proper SVG icon.

Since we will be getting data through the user’s interaction (i.e through the input field), this component will, therefore, hold state. For that reason, it will be a class-based component.

Don’t forget, it can be a functional component if we use React Hooks. Coming to that later in the series.

Keep reading!

Also, notice how we are extending the Component class in the React library. Unlike the previous class components where we used React.Component. Here, we are using Component after importing it from the react module like this:

import React, { Component } from "react";

Now, you are exposed to different methods of using a class component.

Ok. Let’s move on.

To use this component in our application, we will import it inside the TodoContainer.js file using this code:

import InputTodo from "./InputTodo"

Then, add <InputTodo /> inside the render() method just below the <Header />. You should have this:

render() {

return (

<div>

<Header />

<InputTodo /> <TodosList

todos={this.state.todos}

handleChangeProps={this.handleChange}

deleteTodoProps={this.delTodo}

/>

</div>

);

}

Save the file. You should have the form fields rendered in the frontend. As we did for the checkbox, we have to make the form input field a controlled field. The first step is to have a state manage the user's input.

So, add this code just above the render() method in the InputTodo component:

state = {

title: ""

};

We can now take this data and assign it to a value prop in the text input tag. So update it so you have:

<input type="text" placeholder="Add todo..." value={this.state.title} />

**Note:** We use checked prop for the input checkbox and value prop for the text input.

Now, the text input field is being controlled by the component state and not the DOM. Hence, you will not be able to write anything in the field because it is assigned a value equal to the current value of the state.

The value is empty as declared in the state object. To change the state, we need to update it through the setState() method.

Again, just like the checked prop, the value prop also exhibits a warning in the console. React is reminding us that we need to add an onChange handler that will keep track of any changes in the field.

So let’s update the text input so it includes this:

onChange={this.onChange}

Then add this code above the render() method:

onChange = e => {

console.log("hello");

};

Don’t forget that we are using this.onChange because the method, onChange() is part of the class.

Save your file. If you try to write in the input field, you’ll see "hello" text being displayed in response to every keystroke inside the console.

Next, we need to handle the event and update the state. Let’s update the onChange method to this:

onChange = e => {

this.setState({ title: e.target.value });};

Save the file. Now you should be able to write something in the input field. You will also see the state of your app in real-time if you open the React tools.

**What’s happening?**

If you type anything inside the input field, the onChange event handler will trigger. This will then call the onChange() class method that will re-render the state using the setState() method.

In the setState() method, we are passing the current value of the state (i.e the input text) to the title using e.target.value.

And if you recall from vanilla JavaScript DOM API, the predefined parameter, e, hold some important information about the event. From there, you can target the specific input field and grab the updated value.

**Handling React form that has more than one text input field**

For instance, if your form requires fields for the name, email and password. First, you would want all those fields included in the state and assigned to them an empty string. After that, you’d have to modify the onChange method to this:

onChange = e => {

this.setState({

[e.target.name]: e.target.value });

};

Then, you add a name attribute to each of the input tags and assign a value with the same name you declared in the state. For instance, in our case, we will have the name="title" included in the text input tag.

If you apply these changes in your code, your InputTodo component should look like this:

import React, { Component } from "react"

class InputTodo extends Component {

state = {

title: "",

}

onChange = e => {

this.setState({

[e.target.name]: e.target.value, })

}

render() {

return (

<form>

<input

type="text"

placeholder="Add todo..."

value={this.state.title}

name="title" onChange={this.onChange}

/>

<button>Submit</button>

</form>

)

}

}

export default InputTodo

With these changes, you can add as many text input fields as you want. Now, instead of having multiple methods to handle different input fields, we modified the setState() method to this:

this.setState({

[e.target.name]: e.target.value,

})

As long as the value of the name attribute in the input tag matches what you have in the state. It will work perfectly.

***Another note:***

Up to this moment, you know what the setState() method is and how to use it. What you are yet to read is that the method merges the object you provide into the current state.

What do I mean?

Let’s say the state in the InputTodo has more than one variable like so:

state = {

fName: "",

lastName: "",

}

And then you update only the first name using the setState() method.

this.setState({

[e.target.name]: e.target.value,

})

React will merge the previous state with that of the update passed to the setState() method.

Meaning, it will replace the first name with the updated value as instructed. But makes sure that the last name is kept intact. It does not override the entire previous state.

You will see a clearer picture when we start working with React Hooks. For now, just take note of that and move on.

**Updating the Todos list**

At the moment, the page will reload if you try to submit a todos item and update the state data. We need to handle that. To submit todos items, we will make use of the onSubmit event handler on the form element.

Let's quickly do that. Update the <form> tag in the InputTodo component to include the onSubmit handler.

<form onSubmit={this.handleSubmit}>

Then, add the following code above the render() method and save the file:

handleSubmit = e => {

e.preventDefault();

console.log(this.state.title);

};

For the meantime, if you submit your todos items, they will be displayed in the Console tab of your browser.

Note how we are preventing the default behaviour of the form submission.

Now, instead of logging the user's input (titles) in the console, we want to pass them from this component to the TodoContainer component and update the state data. To do that, we will need to raise and handle the event just like we have been doing.

But this time, we will raise an event from the InputTodo component (that accept the user’s input) to the parent component, TodoContainer where the state data to be updated live.

If you check the app diagram or open the React Tools to see the component hierarchy, you will see that we are moving up a level from the InputTodo to the TodoContainer component.

Let’s do it again. As usual, let's start by enabling communication between those components. Starting from the parent component, TodoContainer , add this class method above the render() method:

addTodoItem = title => {

console.log(title);

};

Since we are expecting the todos title from the InputTodo component, you have to include it as the function argument as seen in the code above.

Next, pass this class method to the InputTodo component by updating the <InputTodo /> so you have:

<InputTodo addTodoProps={this.addTodoItem} />

Now, the addTodoItem() method can be accessed through props in the InputTodo component. So update the handleSubmit method in the InputTodo component so you have:

handleSubmit = e => {

e.preventDefault();

this.props.addTodoProps(this.state.title);};

Save the file. You should still be able to see your todos in the console.

Before we move on, let’s clear the input field once we have submitted a todos item for subsequent entry. Simply update the handleSubmit method so you have:

handleSubmit = e => {

e.preventDefault();

this.props.addTodoProps(this.state.title);

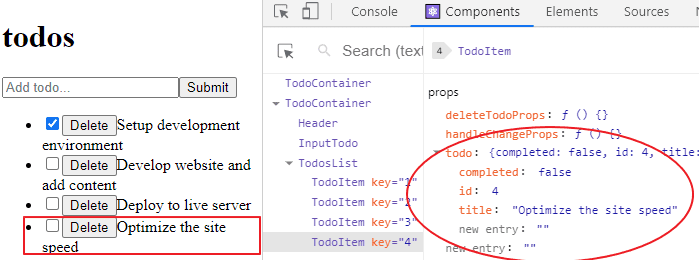
this.setState({ title: "" });};

Finally, we can update the state. Back to the TodoContainer component, update the addTodoItem() method so you have:

addTodoItem = title => {

const newTodo = { id: 4, title: title, completed: false }; this.setState({ todos: [...this.state.todos, newTodo] });};

Save your files. Go to the frontend and add a new todos item to the list.



**What did we do?**

In the code, we started by defining an object for the new item. In this object, we are passing a set of key-value pair. Here, we have the title from the user’s input, the completed key assigned a false value so that the checkbox is not selected by default. Then, for the meantime, we are working with hardcoded id.

With the setState() method, we are re-rendering the state. We are adding the new item to the current todos list which can be grabbed using the spread operator (…).

**Generating random ids for the todos list items**

If you try to submit more than one todos item, React will trigger a console warning telling you that the ids/keys are meant to be unique. This is happening because we are assigning a single id (an id of 4) to every new todos item received through the input field.

That is not ideal. The id help React to identify which item is added or removed.

To fix this warning, we will install something that will generate random ids that we can use. This is called the UUID (Universal Unique Identifier).

To install it, stop the server with CTRL + C. From your terminal, run:

C:\Users\Your Name\simple-todo-app > npm i uuid

Once installed, start the server again with npm start.

To use these ids in your app, you need to import the UUID in the TodoContainer.js file. So, add this line below the list of the import statements.

import { v4 as uuidv4 } from "uuid";

Then, replace any hardcoded id value with uuidv4(). For instance, instead of having:

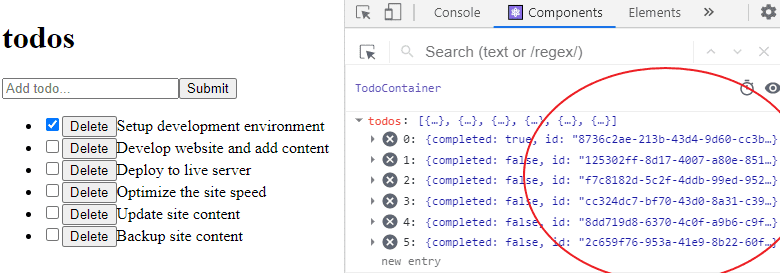
id: 1,

You’ll have:

id: uuidv4(),

Do the same for the other static ids. Save the file and test your application. You should be able to add as many todos items without any console error.

If you check the React tools, you’ll see that the todos items are assigned unique ids from the UUID.



**Validating the Form Submission**

We will keep this simple. Here, we will be checking the text input for empty string before allowing todos submission.

If you are working on a complex project, some libraries make this validation easier. For instance, [the Formik](https://formik.org/) in combination with [the Yup library](https://github.com/jquense/yup).

Formik allows you to create forms without having to manage the form values on your own or write the event handlers for the input.

At this point, you don’t have to worry about the library as it requires a knowledge of the React Hooks to understand how it works.

As suggested by one of the reader, [Andrzej Bakun](https://github.com/anddrzejb), we will be using the JavaScript trim() method.

The trim() method allows us to remove whitespace from both sides of a string. This makes sure that the input field is not empty. Once the condition is satisfied, we update the UI else, we will alert the users to input an item.

So head over to the InputTodo.js file and update the handleSubmit() so you have:

handleSubmit = e => {

e.preventDefault()

if (this.state.title.trim()) {

this.props.addTodoProps(this.state.title)

this.setState({

title: "",

})

} else {

alert("Please write item")

}

}

Great! We are getting there.

Now you know the logic behind form handling in React. Not only that, you now know how to raise and handle events. In the next section, you will learn how to implement CSS in your React application.