

React interactivity: Events and state

[< Previous](#)[↑ Overview: Client-side JavaScript frameworks](#)[Next >](#)

With our component plan worked out, it's now time to start updating our app from a completely static UI to one that actually allows us to interact and change things. In this article we'll do this, digging into events and state along the way, and ending up with an app in which we can successfully add and delete tasks, and toggle tasks as completed.

Prerequisites: Familiarity with the core [HTML](#), [CSS](#), and [JavaScript](#) languages, knowledge of the [terminal/command line](#).

Objective: To learn about handling events and state in React, and use those to start making the case study app interactive.

Handling events

If you've only written vanilla JavaScript before now, you might be used to having a separate JavaScript file, where you query for some DOM nodes and attach listeners to them. For example:

```
1 const btn = document.querySelector('button');
2
3 btn.addEventListener('click', () => {
4   alert("hi!");
5 });
```

In React, we write event handlers directly on the elements in our JSX, like this:

```
1 <button
2   type="button"
3   onClick={() => alert("hi!")}
4 >
5   Say hi!
6 </button>
```

Note: This may seem counter-intuitive regarding best-practice advice that tends to advise against use of inline event handlers on HTML, but remember that JSX is actually part of your JavaScript.

In the above example, we're adding an `onClick` attribute to the `<button>` element. The value of that attribute is a function that triggers a simple alert.

The `onClick` attribute has special meaning here: it tells React to run a given function when the user clicks on the button. There are a couple of other things to note:

- The camel-cased nature of `onClick` is important — JSX will not recognize `onclick` (again, it is already used in JavaScript for a specific purpose, which is related but different — standard `onclick` handler properties).
- All browser events follow this format in JSX — `on`, followed by the name of the event.

Let's apply this to our app, starting in the `Form.js` component.

Handling form submission

At the top of the `Form()` component function, create a function named `handleSubmit()`. This function should [prevent the default behavior of the submit event](#). After that, it should trigger an `alert()`, which can say whatever you'd like. It should end up looking something like this:

```
1 function handleSubmit(e) {  
2   e.preventDefault();  
3   alert('Hello, world!');  
4 }
```

To use this function, add an `onSubmit` attribute to the `<form>` element, and set its value to the `handleSubmit` function:

```
1 | <form onSubmit={handleSubmit}>
```

Now if you head back to your browser and click on the "Add" button, your browser will show you an alert dialog with the words "Hello, world!" — or whatever you chose to write there.

Callback props

In React applications, interactivity is rarely confined to just one component: events that happen in one component will affect other parts of the app. When we start giving ourselves the power to make new tasks, things that happen in the `<Form />` component will affect the list rendered in `<App />`.

We want our `handleSubmit()` function to ultimately help us create a new task, so we need a way to pass information from `<Form />` to `<App />`. We can't pass data from child to parent in the same way as we pass data from parent to child using standard props. Instead, we can write a function in `<App />` that will expect some data from our form as an input, then pass

that function to `<Form />` as a prop. This function-as-a-prop is called a callback prop. Once we have our callback prop, we can call it inside `<Form />` to send the right data to `<App />`.

Handling form submission via callbacks

Inside the top of our `App()` component function, create a function named `addTask()` which has a single parameter of `name`:

```
1 function addTask(name) {  
2   alert(name);  
3 }
```

Next, we'll pass `addTask()` into `<Form />` as a prop. The prop can have whatever name you want, but pick a name you'll understand later. Something like `addTask` works, because it matches the name of the function as well as what the function will do. Your `<Form />` component call should be updated as follows:

```
1 <Form addTask={addTask} />
```

Finally, you can use this prop inside the `handleSubmit()` function in your `<Form />` component! Update it as follows:

```
1 function handleSubmit(e) {  
2   e.preventDefault();  
3   props.addTask("Say hello!");  
4 }
```

Clicking on the "Add" button in your browser will prove that the `addTask()` callback function works, but it'd be nice if we could get the alert to show us what we're typing in our input field! This is what we'll do next.

Note: We decided to name our callback prop `addTask` to make it easy to understand what the prop will do. Another common convention you may well come across in React code is to prefix callback prop names with the word `on`, followed by the name of the event that will cause them to be run. For instance, we could have given our form a prop of `onSubmit` with the value of `addTask`.

State and the `useState` hook

So far, we've used props to pass data through our components and this has served us just fine. Now that we're dealing with user input and data updates, however, we need something more.

For one thing, props come from the parent of a component. Our `<Form />` will not be inheriting a new name for our task; our `<input />` element lives directly inside of `<Form />`, so `<Form />` will be directly responsible for creating that new name. We can't ask `<Form />` to spontaneously create its own props, but we *can* ask it to track some of its own data for us. Data such as this, which a component itself owns, is called **state**. State is another powerful

tool for React because components not only *own* state, but can *update* it later. It's not possible to update the props a component receives; only to read them.

React provides a variety of special functions that allow us to provide new capabilities to components, like state. These functions are called **hooks**, and the `useState` hook, as its name implies, is precisely the one we need in order to give our component some state.

To use a React hook, we need to import it from the `react` module. In `Form.js`, change your very first line so that it reads like this:

```
1 | import React, { useState } from "react";
```

This allows us to import the `useState()` function by itself, and utilize it anywhere in this file.

`useState()` creates a piece of state for a component, and its only parameter determines the *initial value* of that state. It returns two things: the state, and a function that can be used to update the state later.

This is a lot to take in at once, so let's try it out. We're going to make ourselves a `name` state, and a function for updating the `name` state.

Write the following above your `handleSubmit()` function, inside `Form()`:

```
1 | const [name, setName] = useState('Use hooks!');
```

What's going on in this line of code?

- We are setting the initial `name` value as "Use hooks!".
- We are defining a function whose job is to modify `name`, called `setName()`.
- `useState()` returns these two things, so we are using [array destructuring](#) to capture them both in separate variables.

Reading state

You can see the `name` state in action right away. Add a `value` attribute to the form's input, and set its value to `name`. Your browser will render "Use hooks!" inside the input.

```
1 | <input
2 |   type="text"
3 |   id="new-todo-input"
4 |   className="input input__lg"
5 |   name="text"
6 |   autoComplete="off"
7 |   value={name}
8 | />
```

Change "Use hooks!" to an empty string once you're done; this is what we want for our initial state.

```
1 | const [name, setName] = useState('');
```

Reading user input

Before we can change the value of `name`, we need to capture a user's input as they type. For this, we can listen to the `onChange` event. Let's write a `handleChange()` function, and listen for it on the `<input />` tag.

```
1 // near the top of the `Form` component
2 function handleChange(e) {
3   console.log("Typing!");
4 }
5
6 // Down in the return statement
7 <input
8   type="text"
9   id="new-todo-input"
10  className="input input__lg"
11  name="text"
12  autoComplete="off"
13  value={name}
14  onChange={handleChange}
15 />
```

Currently, your input's value will not change as you type, but your browser will log the word "Typing!" to the JavaScript console, so we know our event listener is attached to the input. In order to change the input's value, we have to use our `handleChange()` function to update our `name` state.

To read the contents of the input field as they change, you can access the input's `value` property. We can do this inside `handleChange()` by reading `e.target.value`. `e.target` represents the element that fired the `change` event — that's our input. So, `value` is the text inside it.

You can `console.log()` this value to see it in your browser's console.

```
1 function handleChange(e) {
2   console.log(e.target.value);
3 }
```

Updating state

Logging isn't enough — we want to actually store the updated state of the name as the input value changes! Change the `console.log()` to `setName()`, as shown below:

```
1 function handleChange(e) {
2   setName(e.target.value);
3 }
```

Now we need to change our `handleSubmit()` function so that it calls `props.addTask` with `name` as an argument — remember our callback prop? This will serve to send the task back to the `App` component, so we can add it to our list of tasks at some later date. As a matter of good practice, you should clear the input after your form submits, so we'll call `setName()` again with an empty string to do so:

```

1 function handleSubmit(e) {
2   e.preventDefault();
3   props.addTask(name);
4   setName("");
5 }

```

At last, you can type something into the input field in your browser and click *Add* — whatever you typed will appear in an alert dialog.

Your `Form.js` file should now read like this:

```

1 import React, { useState } from "react";
2
3 function Form(props) {
4   const [name, setName] = useState("");
5
6   function handleChange(e) {
7     setName(e.target.value);
8   }
9
10  function handleSubmit(e) {
11    e.preventDefault();
12    props.addTask(name);
13    setName("");
14  }
15  return (
16    <form onSubmit={handleSubmit}>
17      <h2 className="label-wrapper">
18        <label htmlFor="new-todo-input" className="label__lg">
19          What needs to be done?
20        </label>
21      </h2>
22      <input
23        type="text"
24        id="new-todo-input"
25        className="input input__lg"
26        name="text"
27        autoComplete="off"
28        value={name}
29        onChange={handleChange}
30      />
31      <button type="submit" className="btn btn__primary btn__lg">
32        Add
33      </button>
34    </form>
35  );
36 }
37
38 export default Form;

```

Note: One thing you'll notice is that you are able to submit empty tasks by just pressing the Add button without entering a task name. Can you think of a way to disallow empty tasks from being added? As a hint, you probably need to add some kind of check into the `handleSubmit()` function.

Putting it all together: Adding a task

Now that we've practiced with events, callback props, and hooks we're ready to write functionality that will allow a user to add a new task from their browser.

Tasks as state

Import `useState` into `App.js`, so that we can store our tasks in state — update your `React` import line to the following:

```
1 | import React, { useState } from "react";
```

We want to pass `props.tasks` into the `useState()` hook — this will preserve its initial state. Add the following right at the top of your `App()` function definition:

```
1 | const [tasks, setTasks] = useState(props.tasks);
```

Now, we can change our `taskList` mapping so that it is the result of mapping `tasks`, instead of `props.tasks`. Your `taskList` constant declaration should now look like so:

```
1 | const taskList = tasks.map(task => (  
2 |   <Todo  
3 |     id={task.id}  
4 |     name={task.name}  
5 |     completed={task.completed}  
6 |     key={task.id}  
7 |   />  
8 | )  
9 | );
```

Adding a task

We've now got a `setTasks` hook that we can use in our `addTask()` function to update our list of tasks. There's one problem however: we can't just pass the `name` argument of `addTask()` into `setTasks`, because `tasks` is an array of objects and `name` is a string. If we tried to do this, the array would be replaced with the string.

First of all, we need to put name into an object that has the same structure as our existing tasks. Inside of the `addTask()` function, we will make a `newTask` object to add to the array.

We then need to make a new array with this new task added to it and then update the state of the tasks data to this new state. To do this, we can use spread syntax to [copy the existing array](#), and add our object at the end. We then pass this array into `setTasks()` to update the state.

Putting that all together, your `addTask()` function should read like so:

```
1 | function addTask(name) {  
2 |   const newTask = { id: "id", name: name, completed: false };
```

```
3 |   setTasks([...tasks, newTask]);
4 | }
```


Now you can use the browser to add a task to our data! Type anything into the form and click "Add" (or press the `Enter` key) and you'll see your new todo item appear in the UI!

However, we have another problem: our `addTask()` function is giving each task the same `id`. This is bad for accessibility, and makes it impossible for React to tell future tasks apart with the `key` prop. In fact, React will give you a warning in your DevTools console — "Warning: Encountered two children with the same key..."

We need to fix this. Making unique identifiers is a hard problem — one for which the JavaScript community has written some helpful libraries. We'll use `nanoid` because it's tiny, and it works.

Make sure you're in the root directory of your application and run the following terminal command:

```
1 | npm install nanoid
```

 **Note:** If you're using yarn, you'll need the following instead: `yarn add nanoid`

Now we can import `nanoid` into the top of `App.js` so we can use it to create unique IDs for our new tasks. First of all, include the following import line at the top of `App.js`:

```
1 | import { nanoid } from "nanoid";
```

Now let's update `addTask()` so that each task ID becomes a prefix `todo-` plus a unique string generated by `nanoid`. Update your `newTask` constant declaration to this:

```
1 | const newTask = { id: "todo-" + nanoid(), name: name, completed: false };
```

Save everything, and try your app again — now you can add tasks without getting that warning about duplicate IDs.

Detour: counting tasks

Now that we can add new tasks, you may notice a problem: our heading reads 3 tasks remaining, no matter how many tasks we have! We can fix this by counting the length of `taskList` and changing the text of our heading accordingly.

Add this inside your `App()` definition, before the return statement:

```
1 | const headingText = `${taskList.length} tasks remaining`;
```

Hrm. This is almost right, except that if our list ever contains a single task, the heading will still use the word "tasks". We can make this a variable, too. Update the code you just added as follows:


```
1 | const tasksNoun = taskList.length !== 1 ? 'tasks' : 'task';
2 | const headingText = `${taskList.length} ${tasksNoun} remaining`;
```

Now you can replace the list heading's text content with the `headingText` variable. Update your `<h2>` like so:

```
1 | <h2 id="list-heading">{headingText}</h2>
```

Completing a task

You might notice that, when you click on a checkbox, it checks and unchecks appropriately. As a feature of HTML, the browser knows how to remember which checkbox inputs are checked or unchecked without our help. This feature hides a problem, however: toggling a checkbox doesn't change the state in our React application. This means that the browser and our app are now out-of-sync. We have to write our own code to put the browser back in sync with our app.

Proving the bug

Before we fix the problem, let's observe it happening.

We'll start by writing a `toggleTaskCompleted()` function in our `App()` component. This function will have an `id` parameter, but we're not going to use it yet. For now, we'll log the first task in the array to the console – we're going to inspect what happens when we check or uncheck it in our browser:

Add this just above your `taskList` constant declaration:

```
1 | function toggleTaskCompleted(id) {
2 |   console.log(tasks[0])
3 | }
```

Next, we'll add `toggleTaskCompleted` to the props of each `<Todo/>` component rendered inside our `taskList`; update it like so:

```
1 | const taskList = tasks.map(task => (
2 |   <Todo
3 |     id={task.id}
4 |     name={task.name}
5 |     completed={task.completed}
6 |     key={task.id}
7 |     toggleTaskCompleted={toggleTaskCompleted}
8 |   />
9 | ));
```

Next, go over to your `Todo.js` component and add an `onChange` handler to your `<input />` element, which should use an anonymous function to call

`props.toggleTaskCompleted()` with a parameter of `props.id`. The `<input />` should now look like this:

```
1 | <input
2 |   id={props.id}
3 |   type="checkbox"
4 |   defaultChecked={props.completed}
5 |   onChange={() => props.toggleTaskCompleted(props.id)}
6 | />
```

Save everything and return to your browser and notice that our first task, Eat, is checked. Open your JavaScript console, then click on the checkbox next to Eat. It unchecks, as we expect. Your JavaScript console, however, will log something like this:

```
1 | Object { id: "task-0", name: "Eat", completed: true }
```

The checkbox unchecks in the browser, but our console tells us that Eat is still completed. We will fix that next!

Synchronizing the browser with our data

Let's revisit our `toggleTaskCompleted()` function in `App.js`. We want it to change the `completed` property of only the task that was toggled, and leave all the others alone. To do this, we'll `map()` over the task list and just change the one we completed.

Update your `toggleTaskCompleted()` function to the following:

```
1 | function toggleTaskCompleted(id) {
2 |   const updatedTasks = tasks.map(task => {
3 |     // if this task has the same ID as the edited task
4 |     if (id === task.id) {
5 |       // use object spread to make a new object
6 |       // whose `completed` prop has been inverted
7 |       return {...task, completed: !task.completed}
8 |     }
9 |     return task;
10 |   });
11 |   setTasks(updatedTasks);
12 | }
```

Here, we define an `updatedTasks` constant that maps over the original `tasks` array. If the task's `id` property matches the `id` provided to the function, we use [object spread syntax](#) to create a new object, and toggle the `checked` property of that object before returning it. If it doesn't match, we return the original object.

Then we call `setTasks()` with this new array in order to update our state.

Deleting a task

Deleting a task will follow a similar pattern to toggling its completed state: We need to define a function for updating our state, then pass that function into `<Todo/>` as a prop and call it when the right event happens.

The `deleteTask` callback prop

Here we'll start by writing a `deleteTask()` function in your `App` component. Like `toggleTaskCompleted()`, this function will take an `id` parameter, and we will log that `id` to the console to start with. Add the following below `toggleTaskCompleted()`:

```
1 function deleteTask(id) {  
2   console.log(id)  
3 }
```

Next, add another callback prop to our array of `<Todo />` components:

```
1 const taskList = tasks.map(task => (  
2   <Todo  
3     id={task.id}  
4     name={task.name}  
5     completed={task.completed}  
6     key={task.id}  
7     toggleTaskCompleted={toggleTaskCompleted}  
8     deleteTask={deleteTask}  
9   />  
10 ));
```

In `Todo.js`, we want to call `props.deleteTask()` when the "Delete" button is pressed. `deleteTask()` needs to know the ID of the task that called it, so it can delete the correct task from the state

Update the "Delete" button inside `Todo.js`, like so:

```
1 <button  
2   type="button"  
3   className="btn btn__danger"  
4   onClick={() => props.deleteTask(props.id)}  
5 >  
6   Delete <span className="visually-hidden">{props.name}</span>  
7 </button>
```

Now when you click on any of the "Delete" buttons in the app, your browser console should log the ID of the related task.

Deleting tasks from state and UI

Now that we know `deleteTask()` is invoked correctly, we can call our `setTasks()` hook in `deleteTask()` to actually delete that task from the app's state as well as visually in the app UI. Since `setTasks()` expects an array as an argument, we should provide it with a new

array that copies the existing tasks, *excluding* the task whose ID matches the one passed into `deleteTask()`.

This is a perfect opportunity to use `Array.prototype.filter()`. We can test each task, and exclude a task from the new array if its `id` prop matches the `id` parameter passed into `deleteTask()`.

Update the `deleteTask()` function inside your `App.js` file as follows:

```
1 function deleteTask(id) {  
2   const remainingTasks = tasks.filter(task => id !== task.id);  
3   setTasks(remainingTasks);  
4 }
```

Try your app out again. Now you should be able to delete a task from your app!

Summary

That's enough for one article. Here we've given you the lowdown on how React deals with events and handles state, and implemented functionality to add tasks, delete tasks, and toggle tasks as completed. We are nearly there. In the next article we'll implement functionality to edit existing tasks and filter the list of tasks between all, completed, and incomplete tasks. We'll look at conditional UI rendering along the way.

[< Previous](#)[↑ Overview: Client-side JavaScript frameworks](#)[Next >](#)

In this module

- [Introduction to client-side frameworks](#)
- [Framework main features](#)
- [React](#)
 - [Getting started with React](#)
 - [Beginning our React todo list](#)
 - [Componentizing our React app](#)
 - [React interactivity: Events and state](#)
 - [React interactivity: Editing, filtering, conditional rendering](#)
 - [Accessibility in React](#)
 - [React resources](#)
- [Ember](#)
 - [Getting started with Ember](#)
 - [Ember app structure and componentization](#)
 - [Ember interactivity: Events, classes and state](#)

- [Ember Interactivity: Footer functionality, conditional rendering](#)
- [Routing in Ember](#)
- [Ember resources and troubleshooting](#)
- [Vue](#)
 - [Getting started with Vue](#)
 - [Creating our first Vue component](#)
 - [Rendering a list of Vue components](#)
 - [Adding a new todo form: Vue events, methods, and models](#)
 - [Styling Vue components with CSS](#)
 - [Using Vue computed properties](#)
 - [Vue conditional rendering: editing existing todos](#)
 - [Focus management with Vue refs](#)
 - [Vue resources](#)
- [Svelte](#)
 - [Getting started with Svelte](#)
 - [Starting our Svelte Todo list app](#)
 - [Dynamic behavior in Svelte: working with variables and props](#)
 - [Componentizing our Svelte app](#)
 - [Advanced Svelte: Reactivity, lifecycle, accessibility](#)
 - [Working with Svelte stores](#)
 - [TypeScript support in Svelte](#)
 - [Deployment and next steps](#)

 **Last modified:** Sep 1, 2020, by MDN contributors

Related Topics

[Complete beginners start here!](#)

- ▶ [Getting started with the Web](#)

[HTML — Structuring the Web](#)

- ▶ [Introduction to HTML](#)
- ▶ [Multimedia and embedding](#)
- ▶ [HTML tables](#)

[CSS — Styling the Web](#)

- ▶ [CSS first steps](#)
- ▶ [CSS building blocks](#)
- ▶ [Styling text](#)
- ▶ [CSS layout](#)

[JavaScript — Dynamic client-side scripting](#)

- ▶ [JavaScript first steps](#)
- ▶ [JavaScript building blocks](#)
- ▶ [Introducing JavaScript objects](#)
- ▶ [Asynchronous JavaScript](#)
- ▶ [Client-side web APIs](#)

Web forms — Working with user data

- ▶ [Core forms learning pathway](#)
- ▶ [Advanced forms articles](#)

Accessibility — Make the web usable by everyone

- ▶ [Accessibility guides](#)
- ▶ [Accessibility assessment](#)

Tools and testing

- ▶ [Client-side web development tools](#)
- ▼ [Introduction to client-side frameworks](#)

[Client-side frameworks overview](#)

[Framework main features](#)

▼ [React](#)

[Getting started with React](#)

[Beginning our React todo list](#)

[Componentizing our React app](#)

[React interactivity: Events and state](#)

[React interactivity: Editing, filtering, conditional rendering](#)

[Accessibility in React](#)

[React resources](#)

▼ [Ember](#)

[Getting started with Ember](#)

[Ember app structure and componentization](#)

[Ember interactivity: Events, classes and state](#)

[Ember Interactivity: Footer functionality, conditional rendering](#)

[Routing in Ember](#)

[Ember resources and troubleshooting](#)

▼ [Vue](#)

[Getting started with Vue](#)

[Creating our first Vue component](#)

[Rendering a list of Vue components](#)

[Adding a new todo form: Vue events, methods, and models](#)

[Styling Vue components with CSS](#)

[Using Vue computed properties](#)

[Vue conditional rendering: editing existing todos](#)

[Focus management with Vue refs](#)

[Vue resources](#)

► [Git and GitHub](#)

► [Cross browser testing](#)

Server-side website programming

► [First steps](#)

► [Django web framework \(Python\)](#)

► [Express Web Framework \(node.js/JavaScript\)](#)

Further resources

► [Common questions](#)

[How to contribute](#)



Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

Sign up now