React interactivity: Events and state

- Handling events
  - Handling form submission
- Callback props
  - Handling form submission via callbacks
  - Aside: a note on naming conventions
- Persisting and changing data with state
  - Reading state
  - Reading user input
  - Updating state
- Putting it all together: Adding a task
  - Tasks as state
  - Adding a task
- Detour: counting tasks
- Completing a task
  - Proving the bug
  - Synchronizing the browser with our data
- Deleting a task
  - The deleteTask callback prop
- Deleting tasks from state and UI
- Summary

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, and the terminal/command line.
Learning outcomes:	Handling events and state in React, and using 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 in which you query for some DOM nodes and attach listeners to them. For example, an HTML file might have a button in it, like this:

```
<button type="button">Say hi!</button>
```

And a JavaScript file might have some code like this:

```
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  alert("hi!");
});
```

In JSX, the code that describes the UI lives right alongside our event listeners:

```
<button type="button" onClick={() => alert("hi!")}>
  Say hi!
</button>
```

In this example, we're adding an `onClick` attribute to the button element. The value of that attribute is a function that triggers an alert. This may seem counter to best practice advice about not writing event listeners in HTML, but remember: JSX is not HTML.

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.jsx` component.

Handling form submission

At the top of the `Form()` component function (i.e., just below the `function Form() {` line), 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:

```
function handleSubmit(event) {
  event.preventDefault();
  alert("Hello, world!");
}
```

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

```
<form onSubmit={handleSubmit}>{/* … */}</form>
```

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 `App()` function in `App.jsx`, create a function named `addTask()` which has a single parameter of `name`:

```
function addTask(name) {
  alert(name);
}
```

Next, 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:

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

To use this prop, we must change the signature of the `Form()` function in `Form.jsx` so that it accepts `props` as a parameter:

```
function Form(props) {
  // …
}
```

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

```
function handleSubmit(event) {
  event.preventDefault();
  props.addTask("Say hello!");
}
```

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.

Aside: a note on naming conventions

We passed the `addTask()` function into the `<Form />` component as the prop `addTask` so that the relationship between the `addTask()` *function* and the `addTask` *prop* would remain as clear as possible. Keep in mind, though, that prop names do not *need* to be anything in particular. We could have passed `addTask()` into `<Form />` under any other name, such as this:

```
- <Form addTask={addTask} />
+ <Form onSubmit={addTask} />
```

This would make the `addTask()` function available to the `<Form />` component as the prop `onSubmit`. That prop could be used in `Form.jsx` like this:

```
function handleSubmit(event) {
  event.preventDefault();
- props.addTask("Say hello!");
+ props.onSubmit("Say hello!");
}
```

Here, the `on` prefix tells us that the prop is a callback function; `Submit` is our clue that a submit event will trigger this function.

While callback props often match the names of familiar event handlers, like `onSubmit` or `onClick`, they can be named just about anything that helps make their meaning clear. A hypothetical `<Menu />` component might include a callback function that runs when the menu is opened, as well as a separate callback function that runs when it's closed:

```
<Menu onOpen={() => console.log("Hi!")} onClose={() => console.log("Bye!")} />
```

This `on*` naming convention is very common in the React ecosystem, so keep it in mind as you continue your learning. For the sake of clarity, we're going to stick with `addTask` and similar prop names for the rest of this tutorial. If you changed any prop names while reading this section, be sure to change them back before continuing!

Persisting and changing data with state

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 interactivity, however, we need the ability to create new data, retain it, and update it later. Props are not the right tool for this job because they are immutable — a component cannot change or create its own props.

This is where **state** comes in. If we think of props as a way to communicate between components, we can think of state as a way to give components "memory" – information they can hold onto and update as needed.

React provides a special function for introducing state to a component, aptly named `useState()`.

> **NOTE:** `useState()` is part of a special category of functions called **hooks**, each of which can be used to add new functionality to a component. We'll learn about other hooks later on.

To use `useState()`, we need to import it from the React module. Add the following line to the top of your `Form.jsx` file, above the `Form()` function definition:

```
import { useState } from "react";
```

`useState()` takes a single argument that determines the initial value of the state. This argument can be a string, a number, an array, an object, or any other JavaScript data type. `useState()` returns an array containing two items. The first item is the current value of the state; the second item is a function that can be used to update the state.

Let's create a `name` state. Write the following above your `handleSubmit()` function, inside `Form()`:

```
const [name, setName] = useState("Learn React");
```

Several things are happening in this line of code:

- We are defining a `name` constant with the value `"Learn React"`.
- We are defining a function whose job it is to modify `name`, called `setName()`.
- `useState()` returns these two things in an array, 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 "Learn React" inside the input.

```
<input
  type="text"
  id="new-todo-input"
  className="input input__lg"
  name="text"
  autoComplete="off"
  value={name}
/>
```

Change "Learn React" to an empty string once you're done; this is what we want for our initial state:

```
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 />` element.

```
// near the top of the `Form` component
function handleChange() {
  console.log("Typing!");
}

// …

// Down in the return statement
<input
  type="text"
  id="new-todo-input"
  className="input input__lg"
  name="text"
  autoComplete="off"
  value={name}
  onChange={handleChange}
/>;
```

Currently, our input's value will not change when you try to enter text into it, but your browser will log the word "Typing!" to the JavaScript console, so we know our event listener is attached to the input.

To read the user's keystrokes, we must access the input's `value` property. We can do this by reading the `event` object that `handleChange()` receives when it's called. `event`, in turn, has a `target` property, which represents the element that fired the `change` event. That's our input. So, `event.target.value` is the text inside the input.

You can `console.log()` this value to see it in your browser's console. Try updating the `handleChange()` function as follows, and typing in the input to see the result in your console:

```
function handleChange(event) {
  console.log(event.target.value);
}
```

Updating state

Logging isn't enough — we want to actually store what the user types and render it in the input! Change your `console.log()` call to `setName()`, as shown below:

```
function handleChange(event) {
  setName(event.target.value);
}
```

Now when you type in the input, your keystrokes will fill out the input, as you might expect.

We have one more step: 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 is submitted, so we'll call `setName()` again with an empty string to do so:

```
function handleSubmit(event) {
  event.preventDefault();
  props.addTask(name);
  setName("");
}
```

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.jsx` file should now read like this:

```
import { useState } from "react";

function Form(props) {
  const [name, setName] = useState("");

  function handleChange(event) {
    setName(event.target.value);
  }

  function handleSubmit(event) {
    event.preventDefault();
    props.addTask(name);
    setName("");
  }

  return (
```

```
    <form onSubmit={handleSubmit}>
      <h2 className="label-wrapper">
        <label htmlFor="new-todo-input" className="label__lg">
          What needs to be done?
        </label>
      </h2>
      <input
        type="text"
        id="new-todo-input"
        className="input input__lg"
        name="text"
        autoComplete="off"
        value={name}
        onChange={handleChange}
      />
      <button type="submit" className="btn btn__primary btn__lg">
        Add
      </button>
    </form>
  );
}

export default Form;
```

> **NOTE:** You'll notice 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 prevent this? 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

We need to import `useState` into `App.jsx` so that we can store our tasks in state. Add the following to the top of your `App.jsx` file:

```
import { 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:

```
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:

```
const taskList = tasks?.map((task) => (
  <Todo
    id={task.id}
    name={task.name}
    completed={task.completed}
    key={task.id}
  />
));
```

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:

```
function addTask(name) {
  const newTask = { id: "id", name, completed: false };
  setTasks([...tasks, newTask]);
}
```

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:

```
npm install nanoid
```

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

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

```
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:

```
const newTask = { id: `todo-${nanoid()}`, 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:

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

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:

```
const tasksNoun = taskList.length !== 1 ? "tasks" : "task";
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:

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

Save the file, go back to your browser, and try adding some tasks: the count should now update as expected.

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:

```
function toggleTaskCompleted(id) {
  console.log(tasks[0]);
}
```

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

```
const taskList = tasks.map((task) => (
  <Todo
    id={task.id}
    name={task.name}
    completed={task.completed}
    key={task.id}
    toggleTaskCompleted={toggleTaskCompleted}
  />
));
```

Next, go over to your `Todo.jsx` 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:

```
<input
  id={props.id}
  type="checkbox"
  defaultChecked={props.completed}
  onChange={() => props.toggleTaskCompleted(props.id)}
/>
```

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:

```
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.jsx`. 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:

```
function toggleTaskCompleted(id) {
  const updatedTasks = tasks.map((task) => {
    // if this task has the same ID as the edited task
    if (id === task.id) {
      // use object spread to make a new object
      // whose `completed` prop has been inverted
      return { ...task, completed: !task.completed };
    }
    return task;
  });
  setTasks(updatedTasks);
}
```

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 `completed` 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()`:

```
function deleteTask(id) {
  console.log(id);
}
```

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

```
const taskList = tasks.map((task) => (
  <Todo
    id={task.id}
    name={task.name}
    completed={task.completed}
    key={task.id}
    toggleTaskCompleted={toggleTaskCompleted}
    deleteTask={deleteTask}
  />
));
```

In `Todo.jsx`, 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.jsx`, like so:

```
<button
  type="button"
  className="btn btn__danger"
  onClick={() => props.deleteTask(props.id)}>
  Delete <span className="visually-hidden">{props.name}</span>
</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.

At this point, your `Todo.jsx` file should look like this:

```
function Todo(props) {
  return (
    <li className="todo stack-small">
      <div className="c-cb">
        <input
          id={props.id}
          type="checkbox"
          defaultChecked={props.completed}
          onChange={() => props.toggleTaskCompleted(props.id)}
        />
        <label className="todo-label" htmlFor={props.id}>
          {props.name}
```

```
        </label>
      </div>
      <div className="btn-group">
        <button type="button" className="btn">
          Edit <span className="visually-hidden">{props.name}</span>
        </button>
        <button
          type="button"
          className="btn btn__danger"
          onClick={() => props.deleteTask(props.id)}>
          Delete <span className="visually-hidden">{props.name}</span>
        </button>
      </div>
    </li>
  );
}

export default Todo;
```

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` argument passed into `deleteTask()`.

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

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

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

At this point, your `App.jsx` file should look like this:

```
import { useState } from "react";
import { nanoid } from "nanoid";
import Todo from "./components/Todo";
import Form from "./components/Form";
import FilterButton from "./components/FilterButton";

function App(props) {
  const [tasks, setTasks] = useState(props.tasks);

  function addTask(name) {
    const newTask = { id: `todo-${nanoid()}`, name, completed: false };
    setTasks([...tasks, newTask]);
  }

  function toggleTaskCompleted(id) {
    const updatedTasks = tasks.map((task) => {
      // if this task has the same ID as the edited task
      if (id === task.id) {
        // use object spread to make a new object
        // whose `completed` prop has been inverted
        return { ...task, completed: !task.completed };
      }
      return task;
    });
    setTasks(updatedTasks);
  }

  function deleteTask(id) {
    const remainingTasks = tasks.filter((task) => id !== task.id);
    setTasks(remainingTasks);
  }
  const taskList = tasks?.map((task) => (
    <Todo
      id={task.id}
      name={task.name}
      completed={task.completed}
      key={task.id}
      toggleTaskCompleted={toggleTaskCompleted}
      deleteTask={deleteTask}
```

```
    />
  ));

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

  return (
    <div classNam="todoapp stack-large">
      <h1>TodoMatic</h1>
      <Form addTask={addTask} />
      <div className="filters btn-group stack-exception">
        <FilterButton />
        <FilterButton />
        <FilterButton />
      </div>
      <h2 id="list-heading">{headingText}</h2>
      <ul
        role="list"
        className="todo-list stack-large stack-exception"
        aria-labelledby="list-heading">
        {taskList}
      </ul>
    </div>
  );
}

export default 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.

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:32:38 UTC