



To make Medium work, we log user data.
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.

Open in app

Get started



Published in Capital One Tech



Azat Mardan

Follow

Nov 9, 2017 · 8 min read · Listen

How to Work with Forms, Inputs and Events in React



This is an excerpt from the book [React Quickly](#), available at [manning.com](#)


This article covers how to capture text input and input via other form elements like `<input>`, `<textarea>`, and `<option>`. Working with them is paramount to web development because they allow our applications to receive data (e.g. text) and actions (e.g. clicks) from users.

Recommended Way of Working with Forms in React

In regular HTML, when we work with an input element, the page's DOM maintains that element's value in its DOM node. It's possible to access the value via methods like:





To make Medium work, we log user data.  [Learn more](#)
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.

Open in app

Get started

In React, when working with form elements such as standalone text fields or buttons, developers have an interesting problem to solve. From React's documentation: *"React components must represent the state of the view at any point in time and not only at initialization time."* React is all about keeping things simple by using declarative style to describe UIs. React describes the UI, its end stage, and how it should look.

Can you spot a conflict? In the traditional HTML form elements, the state of the elements will change with the user input. React uses a declarative approach to describe the UI. The input needs to be dynamic to reflect the state properly.

If developers opt NOT to maintain the component state (in JavaScript), nor sync it with the view, then it creates problems — there might be a situation when internal state and view are different and React won't know about changed state. This can lead to all sorts of trouble, and mitigates the simple philosophy of React. The best practice is to keep React's `render()` as close to the real DOM as possible, and that includes the data in the form elements.

Consider this example of a text input field. React must include the new value in its `render()` for that component. Consequently, we need to set the value for our element to new value using `value`. If we implement an `<input>` field as we always did in HTML, React will keep the `render()` in sync with the real DOM. React won't allow users to change the value. Try it yourself. It might feel nuts but it's the appropriate behavior for React!

```
render() {  
  return <input type="text" name="title" value="Mr." />  
}
```

The code above represents the view at any state, and the value will always be "Mr.". With input fields, they must change in response to the user keystrokes. Given these points, let's make the value dynamic.





To make Medium work, we log user data. ×
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.

Open in app

Get started

```
render() {  
  return <input type="text" name="title" value={this.state.title} />  
}
```

What is the value of state? React won't know about users typing in the form elements. Developers need to implement an event handler to capture changes with `onChange`.

```
handleChange(event) {  
  this.setState({title: event.target.value})  
}  
  
render() {  
  return <input type="text" name="title" value={this.state.title}  
    onChange={this.handleChange.bind(this)} />  
}
```

Given these points, the best practice is for developers to implement the following things to sync the internal state with the view (Figure 1):

1. Define elements in `render()` using values from state.
2. Capture changes of a form element using `onChange()` as they happen.
3. Update the internal state in event handler.
4. New values are saved in state and then the view is updated by a new `render()`.





To make Medium work, we log user data.
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.

× [Open in app](#)

Get started

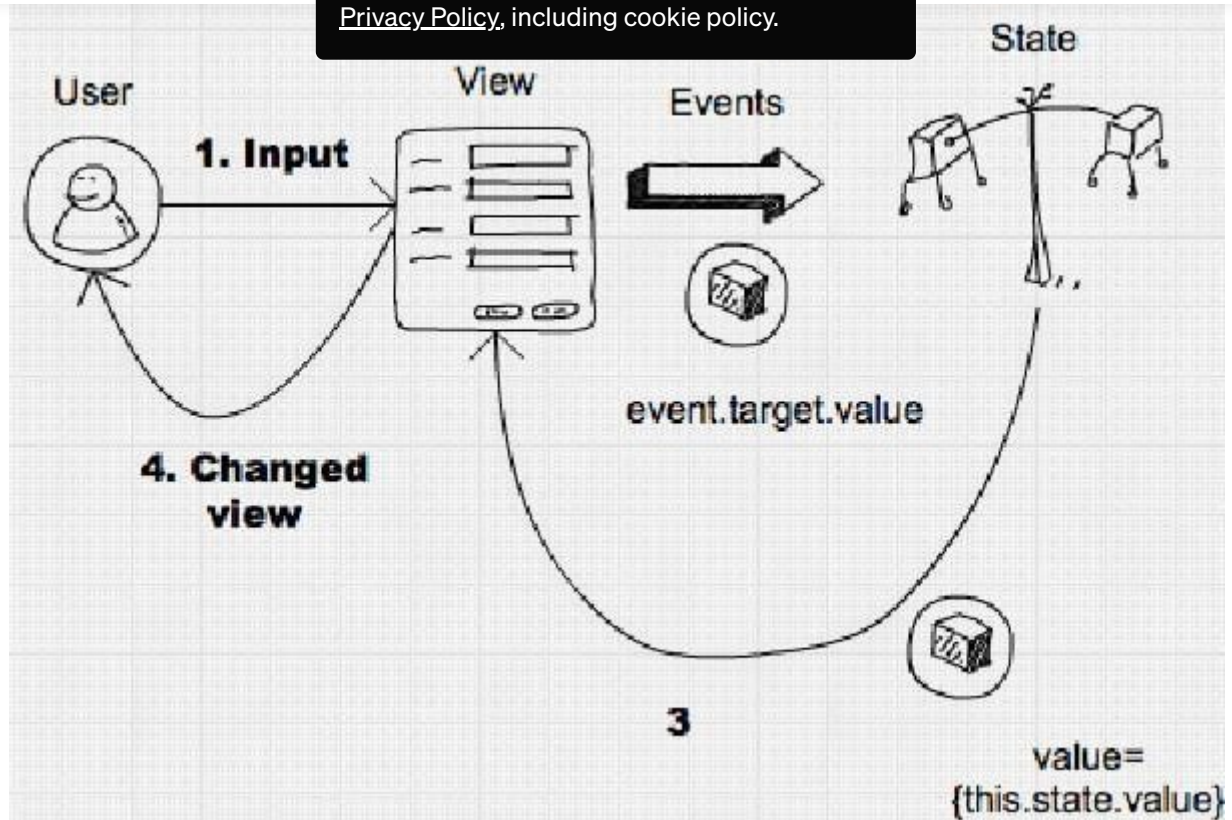


Figure 1: Capturing changes in input and applying to state

It might seem like a lot of work at first glance, but I hope that by using React more, you'll appreciate this approach. It's called a one-way binding because state only changes views. There's no trip back, only a one-way trip from state to view. With one-way binding, a library won't update state (or model) automatically. One of the main benefits of one-way binding is that it removes complexity when working with large apps where many views can implicitly update many states (data models) and vice versa — Figure 2.

Simple doesn't always mean less code. Sometimes, like in this case, developers must write extra code to manually set the data from event handlers to the state (which is rendered to view). But this approach tends to be superior when it comes to complex user interfaces and single-page applications with a myriad of views and states. To put it concisely: *simple isn't always easy*.





To make Medium work, we log user data. [By using Medium, you agree to our Privacy Policy](#), including cookie policy.

Open in app

Get started

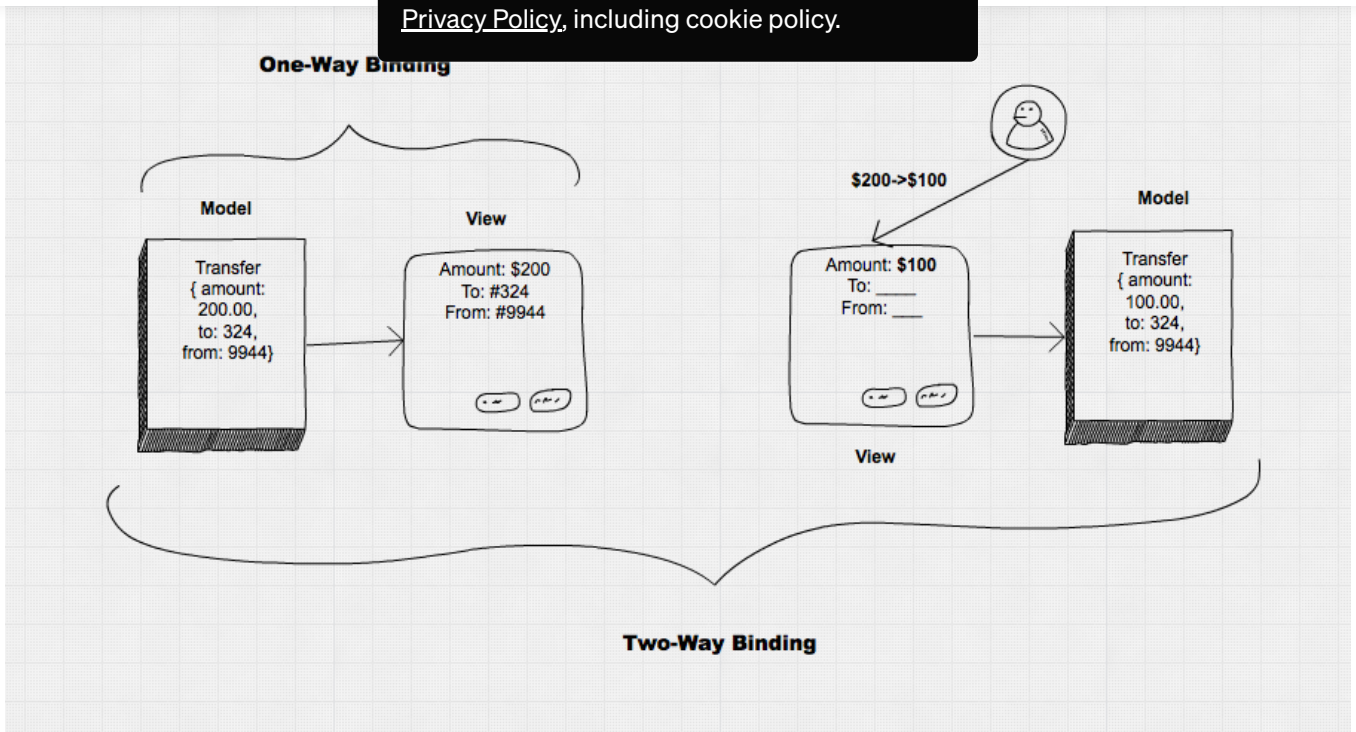


Figure 2: One-way vs two-way binding

Conversely, a two-way binding allows views to change states automatically without developers explicitly implementing it. The two-way binding is how Angular 1 works. Interestingly, Angular 2 borrowed the concept of one-way binding from React and made it the default (you can still have two-way binding explicitly).

For this reason, we'll cover the recommended approach of working with forms first. It's called controlled components and it ensures that the internal component state is always in sync with the view. The alternative approach is uncontrolled component.

So far, we've learned the best practice for working with input fields in React, which is to capture the change and apply it to state as depicted in Figure 1 (input to changed view). Next, let's look at how we define a form and its elements.

Defining Form and its Events in React

We'll start with the `<form>` element. Typically, we don't want our input elements hanging randomly in DOM. This can turn bad if we have many functionally different sets of inputs. Instead, we wrap input elements that share a common purpose in a `<form></form>` element.





To make Medium work, we log user data. ×
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.

Open in app

Get started

each group. React's `<form>` element too. According to [the HTML5 spec](#), developers shouldn't nest forms (it says content is flow content, but with no `<form>` element descendants).

The form element itself can have events. React supports three events for forms in addition to standard React DOM events:

- `onChange` : Fires when there's a change in any of the form's input elements.
- `onInput` : Fires for each change in `<textarea>` and `<input>` elements values. React team doesn't recommend using it (see below).
- `onSubmit` : Fires when the form is submitted, usually by pressing enter.

onChange vs. onInput

React's `onChange` fires on every change in contrast to [the DOM's change event](#), which might not fire on each value change, but fires on lost focus. For example, for `<input type="text">` a user can be typing with no `onChange` and only after the user presses tab or clicks away with his/her mouse to another element (lost focus) will the `onChange` be fired in HTML (regular browser event). As mentioned earlier, in React, `onChange` fires on each keystroke, not only on lost focus.

On the other hand, `onInput` in React is a wrapper for the DOM's `onInput` which fires on each change. Therefore, the React team recommends using `onChange` over `onInput`. The bottom line is that React's `onChange` works differently than `onChange` in HTML, in that it's more consistent (and more like HTML's `onInput`). `onChange` is triggered on every change and not on the loss of focus.

The recommended approach in React is to use `onChange` and `onInput` only when you need to access native behavior for the `onInput` event. The reason is that React's `onChange` wrapper behavior provides consistency.

Using Form Events

In addition to the three events listed above, the `<form>` can have standard React





To make Medium work, we log user data.
By using Medium, you agree to our
[Privacy Policy](#), including cookie policy.



Open in app

Get started

For example, it's good UI to have a form that creates a new line on enter (assuming you're not in the textarea field, in which case enter should create a new line). We can listen to the form submit event by creating an event listener which triggers

```
this.handleSubmit() .
```

```
handleSubmit(event) {  
  
  ...  
  
}  
  
render() {  
  return (  
    <form onSubmit={this.handleSubmit}>  
      <input type="text" name="email" />  
    </form>  
  )  
}
```

Please note: We'll need to implement the handleSubmit function outside of render(), as we'd do with any other event. There's no naming convention which React requires, and you can name the event handler anything you wish as long as it's understandable and consistent. For now, we'll stick with the most popular convention to prefix the event handler with the word "handle" to distinguish it from regular class methods.

As a reminder, don't invoke a method (don't put parenthesis), and don't use double quotes around the curly braces (right way: `EVENT={this.METHOD}`) when setting the event handler. In React, we pass definition of the function, not it's result, and we use curly braces as values of the JSX attributes.

Another way to implement the form submission on enter is by manually listening to key up event (`onKeyUp`) and checking for the key code (13 for enter).





To make Medium work, we log user data. × [Learn more](#)
By using Medium, you agree to our [Privacy Policy](#), including cookie policy.

Open in app

Get started

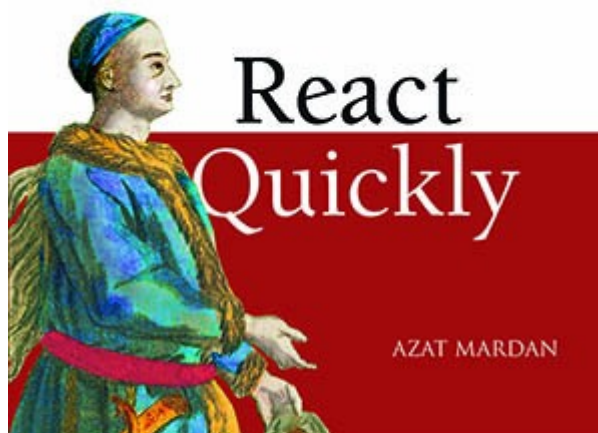
```
render() {  
  return <form onKeyUp={this.handleKeyUp}>  
    ...  
  </form>  
}
```

Please note that the `sendData` method is implemented somewhere else in the code. Also, we'll need to bind(`this`) event handler in `constructor()`.

To summarize, in React we can have events on the form element, not only on individual elements in the form.

Note: The source code for the examples in this article is in the `ch04` folder of the GitHub repository [azat-co/react-quickly](https://github.com/azat-co/react-quickly). And some demos can be found at <http://reactquickly.co/demos>.

That's all for this article, for more on React and its myriad uses check out [React Quickly](https://react.manning.com) at [manning.com](https://react.manning.com).





To make Medium work, we log user data.



By using Medium, you agree to our

[Privacy Policy](#), including cookie policy.

Open in app

Get started

DISCLOSURE STATEMENT

author. Unless noted otherwise in this post, Capital One is not affiliated with, nor is it endorsed by, any of the companies mentioned. All trademarks and other intellectual property used or displayed are the ownership of their respective owners. This article is © 2017 Capital One.

Sign up for Capital One Tech

By Capital One Tech

The low down on our high tech from the engineering experts at Capital One. Learn about the solutions, ideas and stories driving our tech transformation. [Take a look.](#)

Your email



Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.

