API REFERENCE > COMPONENTS >

<input>

The built-in browser `<input>` component lets you render different kinds of form inputs.

```
<input />
```

Reference

`<input>`

To display an input, render the built-in browser `<input>` component.

```
<input name="myInput" />
```

See more examples below.

Props

`<input>` supports all common element props.

You can make an input controlled by passing one of these props:

- `checked`: A boolean. For a checkbox input or a radio button, controls whether it is selected.
- `value`: A string. For a text input, controls its text. (For a radio button, specifies its form data.)

When you pass either of them, you must also pass an `onChange` handler that updates the passed value.

These `<input>` props are only relevant for uncontrolled inputs:

- `defaultChecked`: A boolean. Specifies the initial value for `type="checkbox"` and `type="radio"` inputs.
- `defaultValue`: A string. Specifies the initial value for a text input.

These `<input>` props are relevant both for uncontrolled and controlled inputs:

- `accept`: A string. Specifies which filetypes are accepted by a `type="file"` input.
- `alt`: A string. Specifies the alternative image text for a `type="image"`

input.

- `capture`: A string. Specifies the media (microphone, video, or camera) captured by a `type="file"` input.
- `autoComplete`: A string. Specifies one of the possible autocomplete behaviors.
- `autoFocus`: A boolean. If `true`, React will focus the element on mount.
- `dirname`: A string. Specifies the form field name for the element's directionality.
- `disabled`: A boolean. If `true`, the input will not be interactive and will appear dimmed.
- `children`: `<input>` does not accept children.
- `form`: A string. Specifies the `id` of the `<form>` this input belongs to. If omitted, it's the closest parent form.
- `formAction`: A string. Overrides the parent `<form action>` for `type="submit"` and `type="image"`.
- `formEnctype`: A string. Overrides the parent `<form enctype>` for `type="submit"` and `type="image"`.
- `formMethod`: A string. Overrides the parent `<form method>` for `type="submit"` and `type="image"`.
- `formNoValidate`: A string. Overrides the parent `<form noValidate>` for `type="submit"` and `type="image"`.
- `formTarget`: A string. Overrides the parent `<form target>` for `type="submit"` and `type="image"`.
- `height`: A string. Specifies the image height for `type="image"`.
- `list`: A string. Specifies the `id` of the `<datalist>` with the autocomplete options.
- `max`: A number. Specifies the maximum value of numerical and datetime inputs.
- `maxLength`: A number. Specifies the maximum length of text and other inputs.
- `min`: A number. Specifies the minimum value of numerical and datetime inputs.

- `minLength`: A number. Specifies the minimum length of text and other inputs.
- `multiple`: A boolean. Specifies whether multiple values are allowed for `<type="file"` and `type="email"`.
- `name`: A string. Specifies the name for this input that's submitted with the form.
- `onChange`: An `Event` handler function. Required for controlled inputs. Fires immediately when the input's value is changed by the user (for example, it fires on every keystroke). Behaves like the browser `input` event.
- `onChangeCapture`: A version of `onChange` that fires in the capture phase.
- `onInput`: An `Event` handler function. Fires immediately when the value is changed by the user. For historical reasons, in React it is idiomatic to use `onChange` instead which works similarly.
- `onInputCapture`: A version of `onInput` that fires in the capture phase.
- `onInvalid`: An `Event` handler function. Fires if an input fails validation on form submit. Unlike the built-in `invalid` event, the React `onInvalid` event bubbles.
- `onInvalidCapture`: A version of `onInvalid` that fires in the capture phase.
- `onSelect`: An `Event` handler function. Fires after the selection inside the `<input>` changes. React extends the `onSelect` event to also fire for empty selection and on edits (which may affect the selection).
- `onSelectCapture`: A version of `onSelect` that fires in the capture phase.
- `pattern`: A string. Specifies the pattern that the `value` must match.
- `placeholder`: A string. Displayed in a dimmed color when the input value is empty.
- `readOnly`: A boolean. If `true`, the input is not editable by the user.
- `required`: A boolean. If `true`, the value must be provided for the form to submit.
- `size`: A number. Similar to setting width, but the unit depends on the control.
- `src`: A string. Specifies the image source for a `type="image"` input.
- `step`: A positive number or an `'any'` string. Specifies the distance between valid values

between valid values.

- `type` : A string. One of the input types.
- `width` : A string. Specifies the image width for a `type="image"` input.

Caveats

- Checkboxes need `checked` (or `defaultChecked`), not `value` (or `defaultValue`).
- If a text input receives a string `value` prop, it will be treated as controlled.
- If a checkbox or a radio button receives a boolean `checked` prop, it will be treated as controlled.
- An input can't be both controlled and uncontrolled at the same time.
- An input cannot switch between being controlled or uncontrolled over its lifetime.
- Every controlled input needs an `onChange` event handler that synchronously updates its backing value.

Usage

Displaying inputs of different types

To display an input, render an `<input>` component. By default, it will be a text input. You can pass `type="checkbox"` for a checkbox, `type="radio"` for a radio button, or one of the other input types.

App.js Download Reset

```
export default function MyForm() {
  return (
    <>
      <label>
        Text input: <input name="myInput" />
      </label>
      <hr />
      <label>
        Checkbox: <input type="checkbox" name="myCheckbox" />
      </label>
      <hr />
      <p>
        Radio buttons:
        <label>
          <input type="radio" name="myRadio" value="option1" />
          Option 1
        </label>
        <label>
          <input type="radio" name="myRadio" value="option2" />
          Option 2
        </label>
        <label>
          <input type="radio" name="myRadio" value="option3" />
          Option 3
        </label>
      </p>
    </>
  );
}

```

Show less

Providing a label for an input

Typically, you will place every `<input>` inside a `<label>` tag. This tells the browser that this label is associated with that input. When the user clicks the label, the browser will automatically focus the input. It's also essential for accessibility: a screen reader will announce the label caption when the user focuses the associated input.

If you can't nest `<input>` into a `<label>`, associate them by passing the same ID to `<input id>` and `<label htmlFor>`. To avoid conflicts between multiple instances of one component, generate such an ID with `useId`.

App.js Download Reset

```
import { useId } from 'react';

export default function Form() {
  const ageInputId = useId();
  return (
    <>
      <label>
        Your first name:
        <input name="firstName" />
      </label>
      <hr />
      <label htmlFor={ageInputId}>Your age:</label>
      <input id={ageInputId} name="age" type="number" />
    </>
  );
}

```

Show less

Providing an initial value for an input

You can optionally specify the initial value for any input. Pass it as the `defaultValue` string for text inputs. Checkboxes and radio buttons should specify the initial value with the `defaultChecked` boolean instead.

App.js Download Reset

```
export default function MyForm() {
  return (
    <>
      <label>
        Text input: <input name="myInput" defaultValue="Some initial va
      </label>
      <hr />
      <label>
        Checkbox: <input type="checkbox" name="myCheckbox" defaultCheck
      </label>
      <hr />
      <p>
        Radio buttons:
        <label>
          <input type="radio" name="myRadio" value="option1" />
          Option 1
        </label>
        <label>
          <input
            type="radio"
            name="myRadio"
            value="option2"
            defaultChecked={true}
          />
          Option 2
        </label>
        <label>
          <input type="radio" name="myRadio" value="option3" />
          Option 3
        </label>
      </p>
    </>
  );
}

```

Show less

Reading the input values when submitting a form

Add a `<form>` around your inputs with a `<button type="submit">` inside. It will call your `<form onSubmit>` event handler. By default, the browser will send the form data to the current URL and refresh the page. You can override that behavior by calling `e.preventDefault()`. To read the form data, use `new FormData(e.target)`.

App.js Download Reset

```
export default function MyForm() {
  function handleSubmit(e) {
    // Prevent the browser from reloading the page
    e.preventDefault();

    // Read the form data
    const form = e.target;
    const formData = new FormData(form);

    // You can pass formData as a fetch body directly:
    fetch('/some-api', { method: form.method, body: formData });

    // Or you can work with it as a plain object:
    const formJson = Object.fromEntries(formData.entries());
    console.log(formJson);
  }

  return (
    <form method="post" onSubmit={handleSubmit}>
      <label>
        Text input: <input name="myInput" defaultValue="Some initial va
      </label>
      <hr />
      <label>
        Checkbox: <input type="checkbox" name="myCheckbox" defaultCheck
      </label>
      <hr />
      <p>
        Radio buttons:
        <label><input type="radio" name="myRadio" value="option1" /> Op
        <label><input type="radio" name="myRadio" value="option2" defau
        <label><input type="radio" name="myRadio" value="option3" /> Op
      </p>
      <hr />
      <button type="reset">Reset form</button>
      <button type="submit">Submit form</button>
    </form>
  );
}
```

Show less

Note

Give a `name` to every `<input>`, for example `<input name="firstName" defaultValue="Taylor" />`. The `name` you specified will be used as a key in the form data, for example `{ firstName: "Taylor" }`.

Pitfall

By default, *any* `<button>` inside a `<form>` will submit it. This can be surprising! If you have your own custom `Button` React component, consider returning `<button type="button">` instead of `<button>`. Then, to be explicit, use `<button type="submit">` for buttons that *are* supposed to submit the form.

Controlling an input with a state variable

An input like `<input />` is *uncontrolled.* Even if you pass an initial value like `<input defaultValue="Initial text" />`, your JSX only specifies the initial value. It does not control what the value should be right now.

To render a *controlled* input, pass the `value` prop to it (or `checked` for checkboxes and radios). React will force the input to always have the `value` you passed. Typically, you will control an input by declaring a state variable:

```
function Form() {
  const [firstName, setFirstName] = useState(''); // Declare a state varia
  // ...
  return (
    <input
      value={firstName} // ...force the input's value to match the state v
      onChange={e => setFirstName(e.target.value)} // ... and update the s
    />
  );
}
```

A controlled input makes sense if you needed state anyway—for example, to re-render your UI on every edit:

```
function Form() {
  const [firstName, setFirstName] = useState('');
  return (
    <>
      <label>
        First name:
        <input value={firstName} onChange={e => setFirstName(e.target.valu
      </label>
      {firstName !== '' && <p>Your name is {firstName}.</p>}
      ...
```

It's also useful if you want to offer multiple ways to adjust the input state (for example, by clicking a button):

```
function Form() {
  // ...
  const [age, setAge] = useState('');
  const ageAsNumber = Number(age);
  return (
    <>
      <label>
        Age:
        <input
          value={age}
          onChange={e => setAge(e.target.value)}
          type="number"
        />
        <button onClick={() => setAge(ageAsNumber + 10)}>
          Add 10 years
        </button>
```

The `value` you pass to controlled components should not be `undefined` or `null`. If you need the initial value to be empty (such as with the `firstName` field below), initialize your state variable to an empty string (`''`).

App.js Download Reset

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

export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [age, setAge] = useState('20');
  const ageAsNumber = Number(age);
  return (
    <>
      <label>
        First name:
        <input
          value={firstName}
          onChange={e => setFirstName(e.target.value)}
        />
      </label>
      <label>
        Age:
        <input
          value={age}
          onChange={e => setAge(e.target.value)}
          type="number"
        />
        <button onClick={() => setAge(ageAsNumber + 10)}>
          Add 10 years
        </button>
      </label>
      {firstName !== '' &&
        <p>Your name is {firstName}.</p>
      }
      {ageAsNumber > 0 &&
        <p>Your age is {ageAsNumber}.</p>
      }
    </>
  );
}

```

Show less

Pitfall

If you pass `value` without `onChange`, it will be impossible to type into the input. When you control an input by passing some `value` to it, you *force* it to always have the value you passed. So if you pass a state variable as a `value` but forget to update that state variable synchronously during the `onChange` event handler, React will revert the input after every keystroke back to the `value` that you specified.

Optimizing re-rendering on every keystroke

When you use a controlled input, you set the state on every keystroke. If the component containing your state re-renders a large tree, this can get slow. There's a few ways you can optimize re-rendering performance.

For example, suppose you start with a form that re-renders all page content on every keystroke:

```
function App() {
  const [firstName, setFirstName] = useState('');
  return (
    <>
      <form>
        <input value={firstName} onChange={e => setFirstName(e.target.valu
      </form>
      <PageContent />
    </>
  );
}
```

Since `<PageContent />` doesn't rely on the input state, you can move the input state into its own component:

```
function App() {
  return (
    <>
```

```
      <SignupForm />
      <PageContent />
    </>
  );
}

function SignupForm() {
  const [firstName, setFirstName] = useState('');
  return (
    <form>
      <input value={firstName} onChange={e => setFirstName(e.target.value)
    </form>
  );
}
```

This significantly improves performance because now only `SignupForm` re-renders on every keystroke.

If there is no way to avoid re-rendering (for example, if `PageContent` depends on the search input's value), `useDeferredValue` lets you keep the controlled input responsive even in the middle of a large re-render.

Troubleshooting

My text input doesn't update when I type into it

If you render an input with `value` but no `onChange`, you will see an error in the console:

```
// 🔴 Bug: controlled text input with no onChange handler
<input value={something} />
```

Console

```
You provided a value prop to a form field without an
onChange handler. This will render a read-only field. If the
field should be mutable use defaultValue. Otherwise, set
either onChange or readOnly.
```

As the error message suggests, if you only wanted to specify the *initial* value, pass `defaultValue` instead:

```
// ✅ Good: uncontrolled input with an initial value
<input defaultValue={something} />
```

If you want to control this input with a state variable, specify an `onChange` handler:

```
// ✅ Good: controlled input with onChange
<input value={something} onChange={e => setSomething(e.target.value)} />
```

If the value is intentionally read-only, add a `readOnly` prop to suppress the error:

```
// ✅ Good: readonly controlled input without on change
<input value={something} readOnly={true} />
```

My checkbox doesn’t update when I click on it

If you render a checkbox with `checked` but no `onChange`, you will see an error in the console:

```
// 🔴 Bug: controlled checkbox with no onChange handler
<input type="checkbox" checked={something} />
```

Console

```
You provided a checked prop to a form field without an onChange handler. This will render a read-only field. If the field should be mutable use defaultChecked. Otherwise, set either onChange or readOnly.
```

As the error message suggests, if you only wanted to specify the *initial* value, pass `defaultChecked` instead:

```
// ✅ Good: uncontrolled checkbox with an initial value
<input type="checkbox" defaultChecked={something} />
```

If you want to control this checkbox with a state variable, specify an `onChange` handler:

```
// ✅ Good: controlled checkbox with onChange
<input type="checkbox" checked={something} onChange={e => setSomething(e.t
```

Pitfall

You need to read `e.target.checked` rather than `e.target.value` for checkboxes.

If the checkbox is intentionally read-only, add a `readOnly` prop to suppress the error:

```
// ✅ Good: readonly controlled input without on change
<input type="checkbox" checked={something} readOnly={true} />
```

My input caret jumps to the beginning on every keystroke

If you control an input, you must update its state variable to the input's value from the DOM during `onChange`.

You can't update it to something other than `e.target.value` (or `e.target.checked` for checkboxes):

```
function handleChange(e) {
  // 🔴 Bug: updating an input to something other than e.target.value
  setFirstName(e.target.value.toUpperCase());
}
```

You also can't update it asynchronously:

```
function handleChange(e) {
  // 🔴 Bug: updating an input asynchronously
  setTimeout(() => {
    setFirstName(e.target.value);
  }, 100);
}
```

To fix your code, update it synchronously to `e.target.value`:

```
function handleChange(e) {
  // ✅ Updating a controlled input to e.target.value synchronously
  setFirstName(e.target.value);
}
```

If this doesn't fix the problem, it's possible that the input gets removed and re-added from the DOM on every keystroke. This can happen if you're accidentally resetting state on every re-render. For example, this can happen if the input or one of its parents always receives a different `key` attribute, or if you nest component definitions (which is not allowed in React and causes the "inner" component to always be considered a different tree).

I'm getting an error: "A component is changing an uncontrolled input to be controlled"

If you provide a `value` to the component, it must remain a string throughout its lifetime.

You cannot pass `value={undefined}` first and later pass `value="some string"` because React won't know whether you want the component to be uncontrolled or controlled. A controlled component should always receive a string `value`, not `null` or `undefined`.

If your `value` is coming from an API or a state variable, it might be initialized to `null` or `undefined`. In that case, either set it to an empty string (`''`) initially, or pass `value={someValue ?? ''}` to ensure `value` is a string.

Similarly, if you pass `checked` to a checkbox, ensure it's always a boolean.

PREVIOUS

Common (e.g. <div>)

NEXT

<option>

How do you like these docs?

Take our survey!

FACEBOOK

Open Source

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

Acknowledgements

Docs Contributors

Meet the Team

Blog

More

React Native

Privacy

Terms