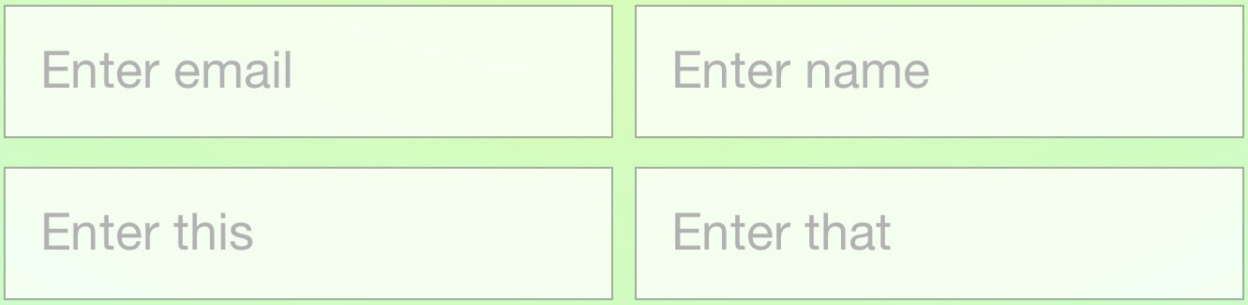
The Missing Forms Handbook of React



by Gosha Arinich

Table of Contents

Hey there!

This is a sample of *The Missing Forms Handbook of React*. It includes chapters 2, *Forms 101*, and 7, *Other cases for forms*. You can get the full book at https://goshakkk.name/the-missing-forms-handbook-of-react/

If anything is not clear, or if you have any questions, I would be more than happy to help. Email me at me@goshakkk.name.

Have a wonderful one!

Gosha Arinich

About you

This book is for you if working with forms in React seems alien. You should already have at least followed the official docs and have a basic idea of how React works.

This book is **not** for you if you are looking for a general React overview. The official docs would be the best place to get started.

About me

I'm Gosha Arinich. I work as a consultant and blog about React.

How to read this book

The Missing Forms Handbook of React is structured as a series of common questions and situations, and answers to them.

The book focuses on the concept of treating forms as data, which is introduced in the first chapters. Later chapters demonstrate practical applications of the concept so that you can "get a feel" for it, as well as cover the specifics of certain cases.

Note that the code snippets are not the point *per se*, they are just a demonstration.

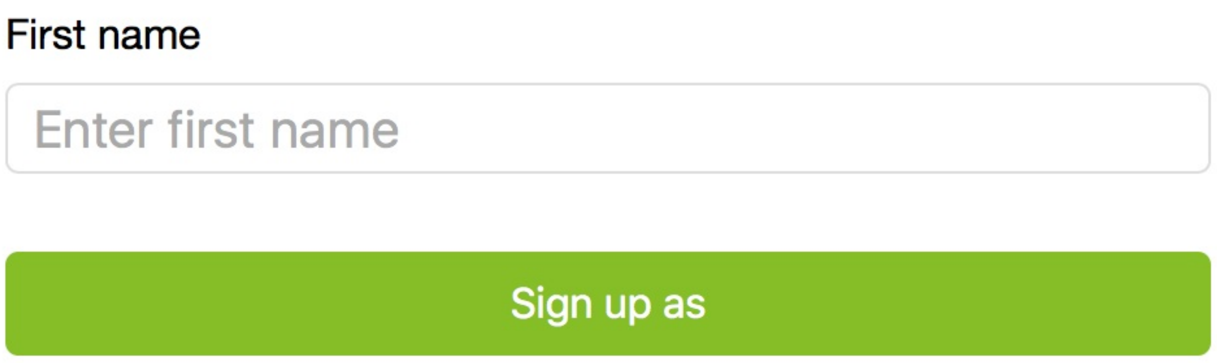
Each chapter is somewhat self-contained so jumping around should be fine if you are looking for the quick fix. However, still read the first couple of chapters for the baseline.

Forms 101

Making the simplest form

We're finally ready to make our first form. Let's dive in.

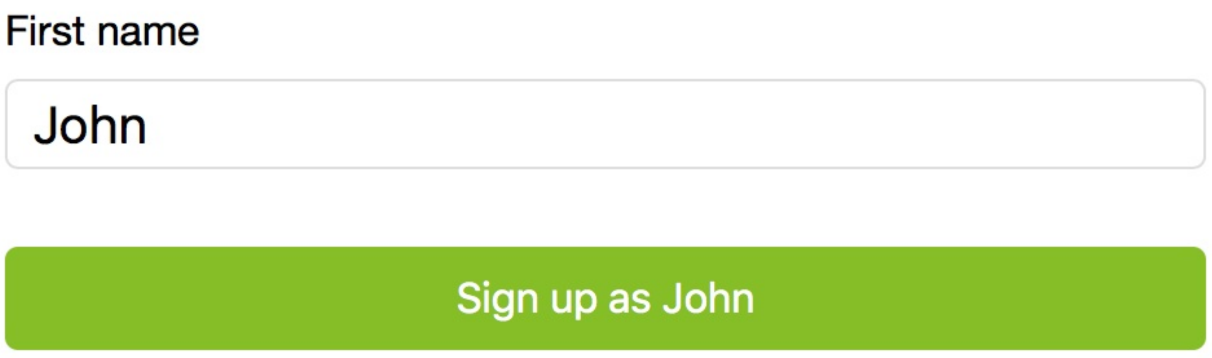
Ready?



```
class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      firstName: '',
    };
  }

  handleFirstNameChange = (evt) => {
    this.setState({ firstName: evt.target.value });
  }

  handleSubmit = (evt) => {
    evt.preventDefault();
    alert(`Signed up as: ${this.state.firstName}`);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          First name
          <input
            type="text"
            placeholder="Enter first name"
            value={this.state.firstName}
            onChange={this.handleFirstNameChange}
          />
        </label>
        <button>Sign up as {this.state.firstName}</button>
      </form>
    )
  }
}
```

🚀 https://goshakkk.jsbin.com/zimituseqo/1/edit?js,output

Did you see *that*? The button label is updated as we type! How cool is that?!



It's stunningly simple to implement this because the form values live in the state.

Now, the code may still look a bit out-of-place, so let's walk through what's going on in here.

Recall what we defined a form as previously:

> - **data**: a collection of fields and their corresponding values. `{`

> `first_name: 'John', last_name: 'Doe' }`
> - **views**: a collection of inputs. `<input value={first_name} onChange={updateFirstName} />`
> - and a way to somehow submit, save, send this data

These are the bits that are absolutely required for any form. There are other optional things, too, but these are present in virtually every form out there. So let's start with them.

Data

The primary purpose of any form is to ask the application user for certain pieces of information. It's no wonder then that this is *the* most important "layer" of any form.

React gives us a mechanism to keep track of, and dynamically change, data within a component. That mechanism is **state**.

We will have a state attribute for every field, and we will pass the current field values to the input components in `render` .

We can say the data of our form is represented by an object of shape `{ firstName: 'Some name' }` .

Updating the Data

A form that doesn't allow changing its inputs is no good. We need a way to update fields!

To make that happen, we're going to write a function that can tie together the input fields and our form's state.

We are going to cover a primitive HTML text input component here, but the same principle applies to other types of inputs and custom input components: whether self-made or from form libraries.

The `<input />` component supports an `onChange` prop, which should be a function. This function is called an *event handler*, because it is going to receive an event as an argument.

`onChange` is going to be called every time the user changes the value of the input.

If you've done any JS/jQuery coding in the past, this is going to feel familiar.

An **event** is an object with many properties. We are only interested in one: `target.value`. `target` is going to be an input DOM node, and `value` holds its new value.

Most commonly, we would update the state according to this new value. Something like this:

```
this.setState({ firstName: evt.target.value` });
```

Aside: getting the value of a checkbox is a bit different. Various input components will be covered later in the book.

Aside: the event for changing a text input value is called `oninput` in the Web APIs. Nevertheless, in React, it is called `onChange` to be in line with other form elements. It also reflects better what's happening — *a value is changing*.

One other thing worth pointing out is the fact we need to bind *event handlers*. The behavior of `this` in JavaScript is a topic deserving its own book, and we're not here to talk about it in great detail.

A quick summary is helpful, so here are three ways it's typically handled:

- **Bind in constructor**. This is the most universal, although a bit verbose approach.

  ```
  class SignUpForm extends React.Component {
    constructor() {
      super();
      this.state = {
        firstName: '',
      };

      this.handleFirstNameChange =
        this.handleFirstNameChange.bind(this);
    }

    handleFirstNameChange(evt) {
      this.setState({ firstName: evt.target.value });
    }
  }
  ```

- **Bind in render**. It's less certainly less verbose than binding in constructor, but it may have performance implications.

  ```
    handleFirstNameChange(evt) {
      this.setState({ firstName: evt.target.value });
    }

    render() {
      // ...
      onChange={this.handleFirstNameChange.bind(this)}
    }
  ```

- **Use an arrow function as a class property**. This syntax is still experimental, but many projects use it.

```
handleFirstNameChange = (evt) => {
  this.setState({ firstName: evt.target.value });
}
```

This style is going to be used throughout the book. If you can't use it on your project, changing this into either bind in constructor or bind in render should get you there.

Relevant material:

Approaches for handling `this`: https://medium.com/@housecor/react-binding-patterns-5-approaches-for-handling-this-92c651b5af56

Why and how to bind methods in your React component classes: http://reactkungfu.com/2015/07/why-and-how-to-bind-methods-in-your-react-component-classes/

View

After with data and update mechanism in place, we need to render the inputs themselves.

We pass in the current value and an event handler from the section before to the input component:

```
<input
  value={this.state.firstName}
  onChange={this.handleFirstNameChange}
/>
```

This is enough to use the primitive `<input />` component, but even with custom UI components, it's still going to be similar.

A way to submit

We also want to respond to the form getting submitted.

To do that, we pass `onSubmit` to the `form` component, and that's it. The "submit" event can be triggered by either:

- clicking a `button` or `input type=submit` in the form
- pressing Enter while in a text box

Which is why you shouldn't add `onClick` to the `button`: it would break the Enter case. Instead, just pass `onSubmit` to the `<form>` directly.

Now, what happens in the submit handler is mostly up to you. You do your thing here, whatever that is — making a request to a remote server or something. In this book, it doesn't matter that much.

One thing you need to make sure is to call `evt.preventDefault()` in the `onSubmit` handler. It's needed to tell the browser not even to try processing this form.

But why do we need to set the state?

It's a super-common question that people ask all the time.

> If I type a letter into the input, wouldn't it just work?

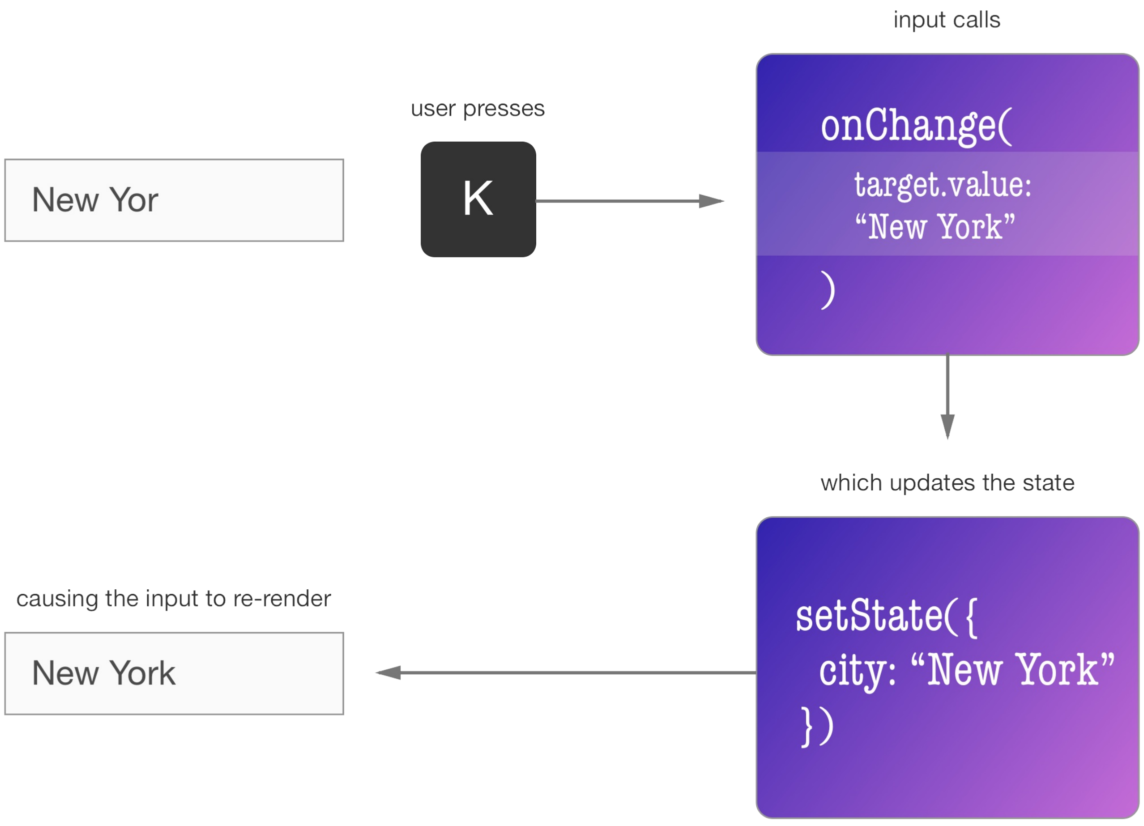
Actually, it won't.

What we've been doing up until now is *controlling* the input. In loose terms, we are setting its value explicitly based on what we have in state. Its value *can't* be different from what we pass into `value={...}`

The only way the string inside the input is going to change is if we update the state... which we do, inside `handleFirstNameChange`.

You may be wondering if the value of the input is always equal to state, wouldn't that mean `evt.target.value` would be that same value? No. The value in the event is what the new value *should* be.

For example, if a name field has the value "New Yor", and you type "k", the event will contain the value "New York". The value from the event is then used to update the state, which changes the value inside input.



🚀 https://goshakkk.jsbin.com/fayupe/2/edit?js,output

However, if you don't update the state inside the event handler, the new value is just not going to stick around, and the field will still say "New Yor".

See for yourself:

🚀 https://goshakkk.jsbin.com/fayupe/4/edit?js,output

How are edits handled?

That's cool, you think, but if the only kind of form were an empty form, everyone would be miserable. Instead, our applications often involve creating and updating data (the C and U parts of CRUD.)

So using forms for editing data is pretty standard. Is there a way to do that in React? Why yes!

When making a form to create, say, a blog post, there is no existing data. But after you create a post, you're more than likely to spot a typo, and therefore would need to *edit* the post.

When you do so, you get a copy of the current text of the post; but as you start changing it, it's not reflected on the blog until you press 'Save.' Same here.

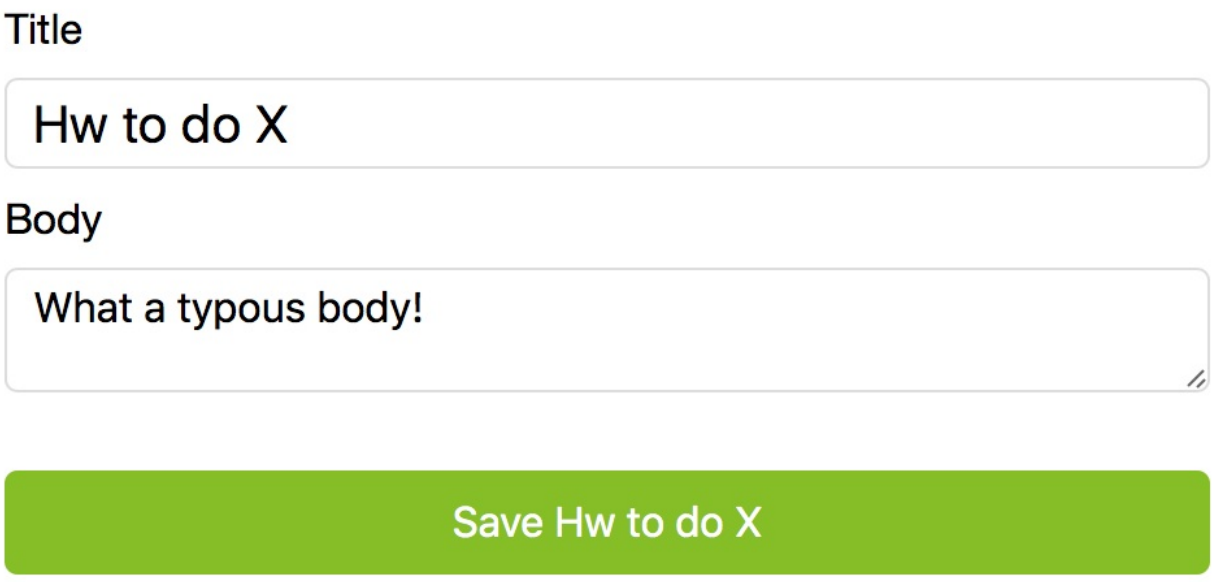
To achieve that, we would simply want to pass existing values to the form component. Like this, perhaps:

```
<PostForm title={currentTitle} body={currentBody} />
```

Finally, this won't be usable unless `PostForm` uses these props to **initialize its state**:

```
class PostForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      title: props.title || '',
      body: props.body || '',
    };
  }
}
```

The code speaks for itself.



🚀 https://goshakkk.jsbin.com/pawicu/4/edit?js,output

Aside: If you read through articles on React, you may see that "setting state from props is an 'anti-pattern'". The main issue is handling updates to the state in response to the props changing. This can be a valid concern but should be understood as advice, not the ultimate rule. In this particular case, this usage makes sense.

Right now, we simply initialize from props.

Other times, it may be needed to respond to props changing. This is not a requirement very often, but if it is, there is just one important thing to think about before implementing: how *exactly* should the form respond to props updating.

Should it update the form state to match the props?

Should it just keep the new props for reference? For example, to implement resetting the form.

Once you determine the desired behavior, the implementation should be straightforward.

Other cases for forms

Not always a string

Sometimes, you have a text input that should be used to enter something other than a string — a number, for example.

But `<input />` expects a string for `value`, and gives one back in `onChange`. What do we do?

This one is pretty easy.

We just need a way to convert between strings and numbers. (Or encode a number into a string, and decode a number from a string.) `parseInt(string, 10)` and `number.toString()` for the win, right?

Mostly. There is just one more case to keep in mind. While an empty value of a string can be represented as `''`, it's different with numbers. Zero is not an "empty number," whatever that means, it's as numbery as it gets.

We should, therefore, have a way to represent the absence of a number — and `null` works perfectly for that.

So, in terms of data, our numerical state field can be either a number or `null`.

To handle it, we are going to need two things:

1. Convert to string to pass into `value={...}`. Easy. If the value is `null`, we're going to pass an empty string, otherwise, `number.toString()`.

2. To convert the string back into a number. Strip non-numerical characters. If the string length is 0, then give a `null`, otherwise `parseInt(x, 10)`.

Also, unlike form logic, this is more than fine to extract into a separate component, just for inputting numbers, or even use an existing one.

Why? Because the form operates in domain values. The fact that the input value is internally a string is an implementation detail that is of no concern to the form itself.

```
const NumberInput = ({ value, onChange, ...rest}) => {
  const stringValue = value != null ? value.toString() : '';
  const handleChange = (evt) => {
    const number = evt.target.value.replace(/\D/g,'');
    onChange(number.length > 0 ? parseInt(number, 10) : null);
  };

  return <input value={stringValue} onChange={handleChange} {...rest
} />;
};
```

🚀 https://goshakkk.jsbin.com/cidage/3/edit?js,output

Note: an input for the decimal number would be slightly more complicated to implement.

Not always a textbox

There is more to forms than just textboxes:

- checkboxes
- radio boxes
- `select` s
- `type=file`

There are quirks with preventing the setting of checkbox/radio value via JS... but React handles these for us 👍

Either way, keep in mind that `touched` isn't valuable for hiding error messages for inputs other than textboxes, and maybe selects.

Checkboxes

Checkboxes are pretty easy to work with. They are the simplest input, one that is passing around either `true` or `false` .

Two minor differences between checkboxes and text inputs are in:

- The name of the prop to pass current value. It's `checked={bool}` instead of `value={someString}` ; and
- The attribute name to get the new value in `onChange` handler is `evt.target.checked` instead of `evt.target.value` .

```
class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      firstName: '',
      agreed: false,
    };
  }

  handleFirstNameChange = (evt) => {
    this.setState({ firstName: evt.target.value });
  }

  handleAgreeChange = (evt) => {
    this.setState({ agreed: evt.target.checked });
  }

  handleSubmit = (evt) => {
    alert(`Signed up as: ${this.state.firstName}`);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type="text"
          placeholder="Enter first name"
          value={this.state.firstName}
          onChange={this.handleFirstNameChange}
        />
        Agree to terms
        <input
          type="checkbox"
          checked={this.state.agreed}
          onChange={this.handleAgreeChange}
        />
        <button>Sign up as {this.state.firstName} ({this.state.agre
ed.toString()})</button>
      </form>
    )
  }
}
```

🚀 https://goshakkk.jsbin.com/xasuka/4/edit?js,output

Radios

Traditionally, radio buttons have been scoped using the `name` attribute:

```
<input
  type="radio"
  name="color"
  value="black"
/>
<input
  type="radio"
  name="color"
  value="green"
/>
```

🚀 https://goshakkk.jsbin.com/veledi/1/edit?html,output

The browser will make sure only one radio with the attribute `name="color"` is selected. The `name` attribute acts as a mechanism to let the browser relate radio buttons together.

In React, however, we have to be more specific, and for a good reason. There's no `name` or anything. Radios are just views that:

- know whether they are selected, and
- can ask the form to change the selection

One thing that makes radios unique is: they usually have both `checked` and `value` props. `checked`, as the name implies, is used to decide whether to mark this button as selected, whereas `value` is often used in the `onChange` handler.

Something along the lines of:

```
<input
  type="radio"
```

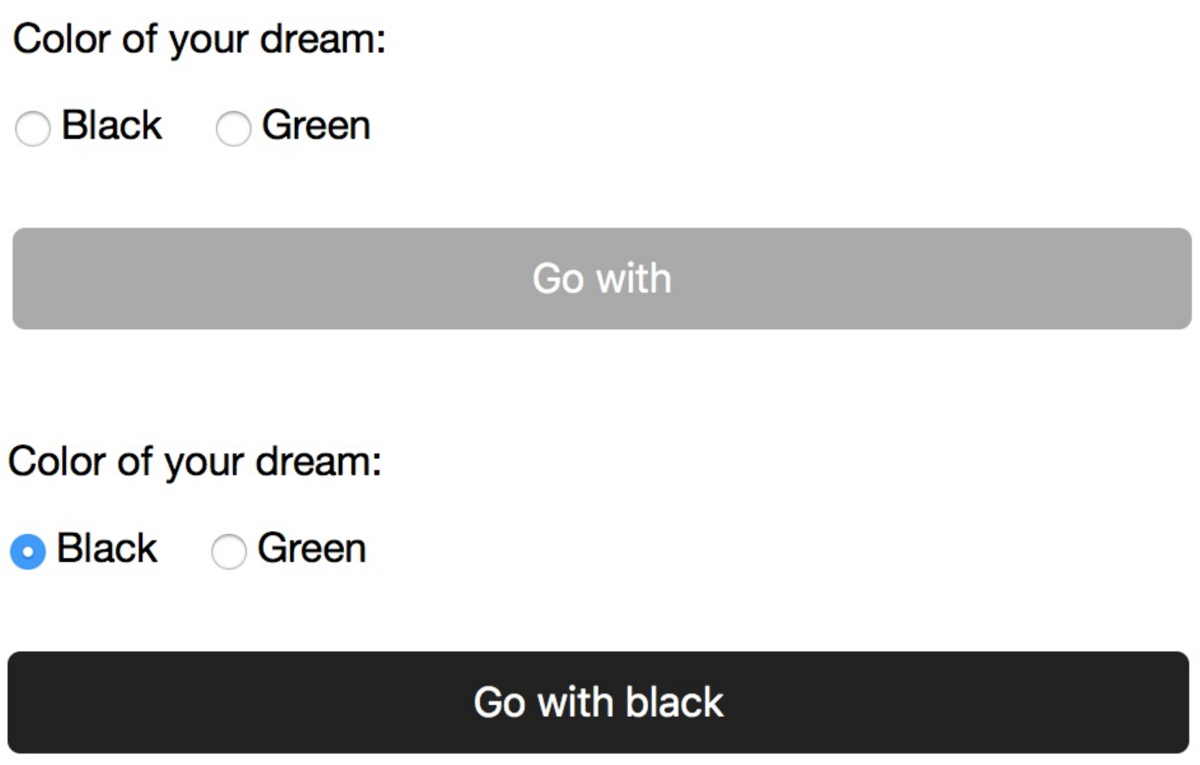
```
    value="black"
    checked={this.state.color === 'black'}
    onChange={this.handleColorChange}
  />
```

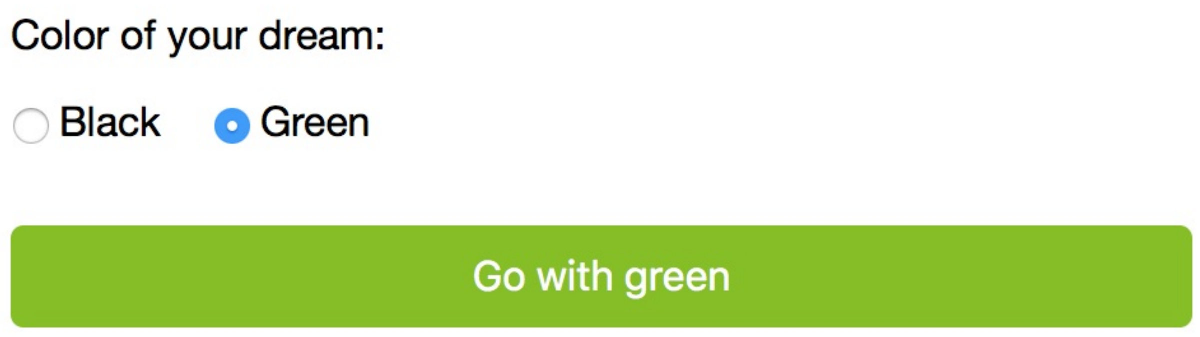
The HTML example from before can be easily translated into React. Observe:

```
class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      color: null,
    };
  }

  handleColorChange = (evt) => {
    this.setState({ color: evt.target.value });
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <p>Color of your dream:</p>
        <label>
          <input
            type="radio"
            value="black"
            checked={this.state.color === 'black'}
            onChange={this.handleColorChange}
          />
          Black
        </label>
        <label>
          <input
            type="radio"
            value="green"
            checked={this.state.color === 'green'}
            onChange={this.handleColorChange}
          />
          Green
        </label>
        <button className={this.state.color}>
          Go with {this.state.color}
        </button>
      </form>
    )
  }
}
```

🚀 https://goshakkk.jsbin.com/rejowa/2/edit?js,output

And look, it can even change the styling based on what's selected:





"Forms are data" at its finest!

Note: even though we are using the `value` attribute to set the new color, there is no reason we couldn't have done it differently.

Instead of relying on `value`, we could generate event handlers on the fly:

```
<input
  type="radio"
  checked={this.state.color === 'black'}
  onChange={this.makeColorChangeHandler('black')}
/>
```

Which is especially handy if you're dealing with data types other than `string` and want to avoid parsing between a string and that other type.

Select

Using select is dead simple...

```
<select value={this.state.color || ""} onChange={this.handleColorCh
ange}>
  <option value="">Choose</option>
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="blue">Blue</option>
</select>
```

Keep in mind that you should always set a `value` prop on the select. Otherwise, the browser may make it look like the first option is selected, but your state will still say nothing is selected — i.e. your state and view will be out of sync.

If the value of select is empty when the form is initialized (`null` in the state), you would need to include an empty `option` , like in the example above.

🚀 https://goshakkk.jsbin.com/yosawa/3/edit?js,output

While a bare `select` can be all you need, a search feature inside a big select is desired. For that, you can use a ready-made component.

File inputs

How else would you update your Twitter pic if it weren't for `<input type="file" />` ?

A file input is somewhat unusual with respect to the data flow... in that, it is not *actually* controlled.

Controlling the value of a file input is not something that makes much sense. The data only flows one way in this case: from the input to the form component, but not both ways.

```
<input type="file" onChange={...} />
```

And in `onChange` , we would store the `File` object (https://developer.mozilla.org/en/docs/Web/API/File) in the state:

```
handleFileChange = (evt) => {
  const file = evt.target.files[0];
  this.setState({ file });
};
```

Sending the `File` to the server is outside the scope of the book, however.

What makes an input controlled?

A form element becomes "controlled" if you set its value via a prop. That's all.

Each of the form elements, though, has a different prop for setting that value, so here's a summary:

Element	Value property	New value in the callback
`<input type="text" />`	`value="string"`	`evt.target.value`
`<textarea />`	`value="string"`	`evt.target.value`
`<input type="checkbox" />`	`checked={boolean}`	`evt.target.checked`
`<input type="radio" />`	`checked={boolean}`	`evt.target.checked`
`<select />`	`value="option value"`	`evt.target.value`

Where should the state go?

When working with more sophisticated forms, it can be the case that the form is shown one part at a time, like a multi-step wizard.

For cases like this, it's important to keep in mind that each individual step should only display the views, but not store the data.

The data should be stored in a common parent of these steps — because that's where it belongs, and that's where it will be used for submission.

My blog post on collecting data from a wizard form demonstrates the idea: https://goshakkk.name/wizard-form-collect-info/

This was a sample of *The Missing Forms Handbook of React*: chapters 2, *Forms 101*, and 7, *Other cases for forms*. You can get the full book at https://goshakkk.name/the-missing-forms-handbook-of-react/

If anything is not clear, or if you have any questions, I would be more than happy to help. Email me at me@goshakkk.name.

Have a wonderful one!

Gosha Arinich