Controlled Components

 (https://github.com/learn-co-curriculum/react-hooks-forms)  (https://github.com/learn-co-curriculum/react-hooks-forms/issues/new)

Learning Goals

- Understand what "controlled components" are in React
- Implement controlled components by synchronizing input values with component state

Introduction

In this lesson, we'll discuss how to set up controlled inputs in React.

If you want to code along, there is starter code in the `src` folder. Make sure to run `npm install && npm start` to see the code in the browser.

> Note: in the examples in this lesson, form submission functionality is omitted for simplicity.

Controlling Form Values From State

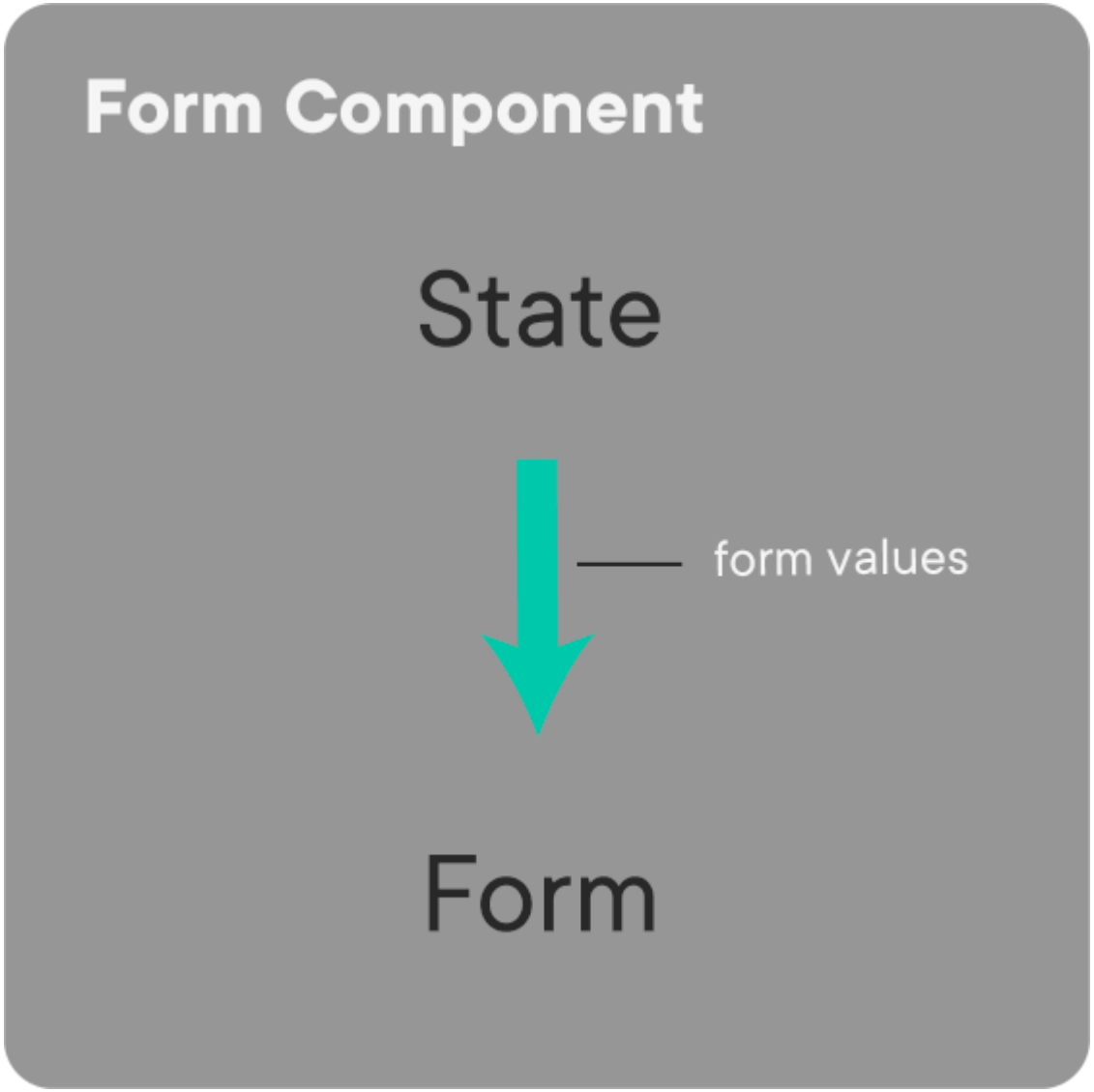
Forms in React are similar to their regular HTML counterparts. The JSX we write is almost identical. The way we store and handle form data, however, is entirely new.

In React, it is often a good idea to set up *controlled* forms. A controlled form is **a form that derives its input values from state**. Consider the following:

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

function Form() {
  const [firstName, setFirstName] = useState("John");
  const [lastName, setLastName] = useState("Henry");
```

```
    return (
      <form>
        <input type="text" value={firstName} />
        <input type="text" value={lastName} />
        <button type="submit">Submit</button>
      </form>
    );
  }

  export default Form;
```

With the setup above, the two text `input` elements will display the corresponding state values.



This code is not quite complete though — as it is now, there is no way to *change* the state. The inputs in the form above will be stuck displaying whatever state is set to.

To completely control a form, we also need our form to *update* state.

Updating State via Forms

If we can change state values, React will re-render and our `input`s will display the new state. We know that `setFirstName` and `setLastName` are what we'll need to initiate a state change, but when would we use them?

We want to fire it **every time the input value changes**. Forms should display whatever changes a user makes, even if it is adding a single letter in an input. For this, we use an event listener, `onChange`, that React has set up for us:

```
<input type="text" onChange={handleFirstNameChange} value={firstName} />
<input type="text" onChange={handleLastNameChange} value={lastName} />
```

We can listen for several types of events on input fields. `onChange` will fire every time the value of an input changes. In our example, we're passing a callback function that accepts `event` as its argument. The `event` data being passed in is automatically provided by the `onChange` event listener. Let's write out what these functions look like:

```
function handleFirstNameChange(event) {
  setFirstName(event.target.value);
}

function handleLastNameChange(event) {
  setLastName(event.target.value);
}
```

The `event` contains data about the `target`, which is whatever DOM element the `event` was triggered on. That `target`, being an `input` element, has a `value` attribute. This attribute is equal to whatever is currently entered into that particular `input`!

Keep in mind, **this is not the value we provided from state**. When we read `event.target.value`, we get whatever content is present when the event fired. In the case of our first input, that would be a combination of whatever `firstName` is equal to *plus* **the last key stroke**. If you pressed 's', `event.target.value` would equal "Johns".

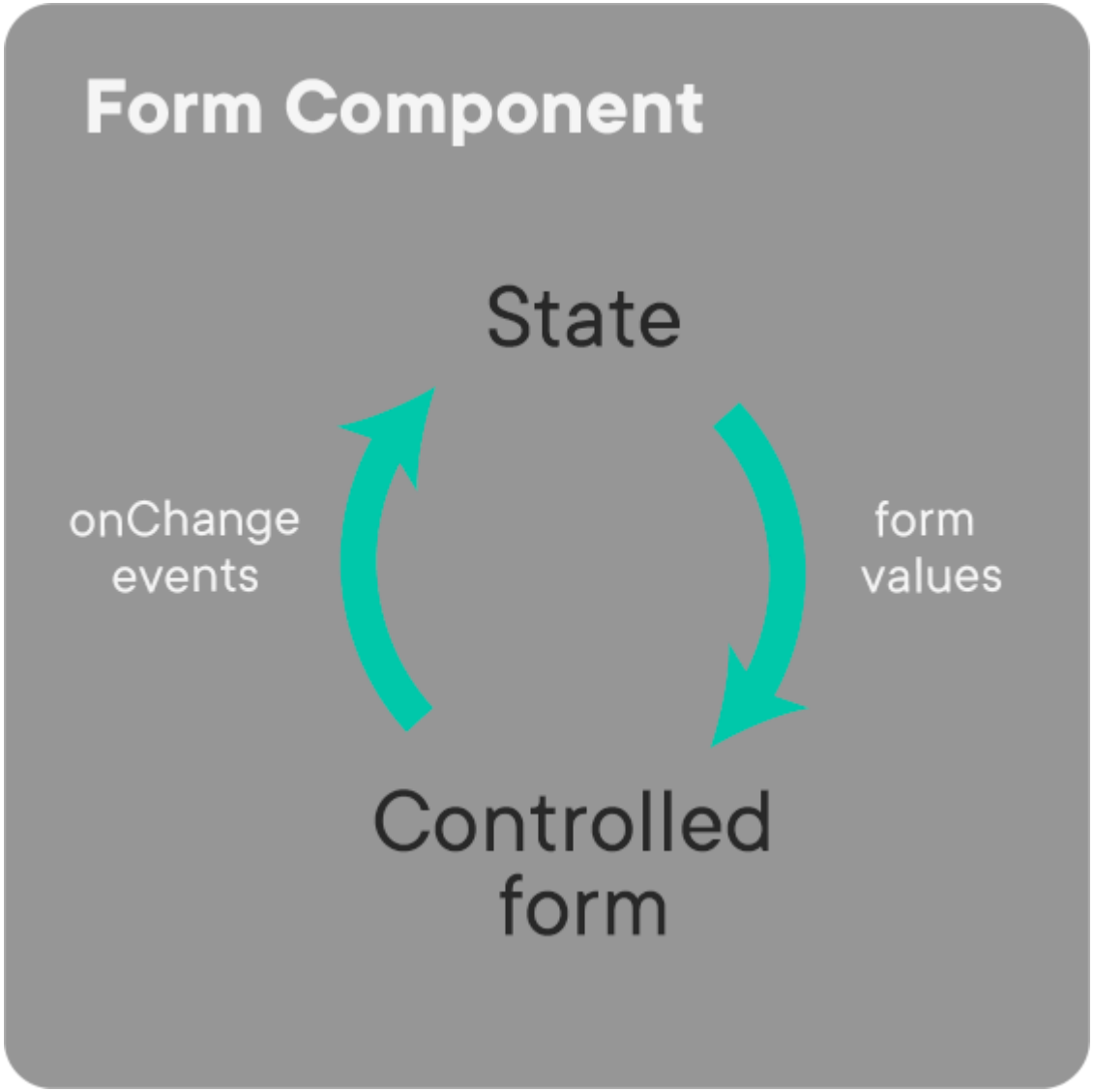
Inside both functions is a function to update state. The two functions are nearly identical, with just one difference: `setFirstName()` changes the `firstName`, and `setLastName()` changes the `lastName`. The full component would look like the following:

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

```
function Form() {
  const [firstName, setFirstName] = useState("John");
  const [lastName, setLastName] = useState("Henry");

  function handleFirstNameChange(event) {
    setFirstName(event.target.value);
  }

  function handleLastNameChange(event) {
    setLastName(event.target.value);
  }

  return (
    <form>
      <input type="text" onChange={handleFirstNameChange} value={firstName} />
      <input type="text" onChange={handleLastNameChange} value={lastName} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default Form;
```

In the `handleFirstNameChange()` and `handleLastNameChange()` functions, we're updating state based on `event.target.value`. This, in turn, causes a re-render... and the cycle completes. The *new* state values we just set are used to set the `value` attributes of our two `input`s.

From a **user**'s perspective, the form behaves exactly how we'd expect, displaying the text that is typed. From **React**'s perspective, we gain control over form values, giving us the ability to more easily manipulate (or restrict) what our `inputs`s display, and send form data to other parts of the app or out onto the internet...



Controlling forms makes it more convenient to share form values between components. Since the form values are stored in state, they are easily passed down as props or sent upward via a function supplied in props.

Form Element Types

Form elements include `<input>`, `<textarea>`, `<select>`, and `<form>` itself. When we talk about inputs in this lesson, we broadly mean the form elements (`<input>`, `<textarea>`, `<select>`) and not always specifically just `<input>`.

To control the value of these inputs, we use a prop specific to that type of input:

- For `<input>`, `<textarea>`, and `<select>`, we use `value`, as we have seen.
- For a checkbox (`<input type="checkbox">`), we use `checked`:

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

function Form() {
  const [newsletter, setNewsletter] = useState(false);

  function handleNewsletterChange(event) {
    // .checked, not .value!
    setNewsletter(event.target.checked);
  }

  return (
    <form>
      <label htmlFor="newsletter">Subscribe to our Newsletter?</label>
      <input
        type="checkbox"
        id="newsletter"
        onChange={handleNewsletterChange}
        {/* checked instead of value */}
        checked={newsletter}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

```
export default Form;
```

Each of the input types has an `onChange` event listener, allowing us to update state when a user interacts with a form. Once that happens, the `value` or `checked` attribute is then set based on the updated state value. Combining these two steps is what enables us to set up controlled forms.

Why Use Controlled Forms When We Do Not Have To

Controlled forms can be very useful for specific purposes — since we can set our state *elsewhere* using this setup, it's easy to populate forms from existing available data.

When we have a controlled form, the state does not need to be stored in the same component. We could store state in a parent component, instead. To demonstrate this, we'll need to create a new component. To keep it simple, we'll call this `ParentComponent`. `ParentComponent` can hold all the functions while `Form` just handles the display of JSX:

```
// src/components/ParentComponent
import React, { useState } from "react";
import Form from "./Form";

function ParentComponent() {
  const [firstName, setFirstName] = useState("John");
  const [lastName, setLastName] = useState("Henry");

  function handleFirstNameChange(event) {
    setFirstName(event.target.value);
  }

  function handleLastNameChange(event) {
    setLastName(event.target.value);
  }
```

```
    return (
      <Form
        firstName={firstName}
        lastName={lastName}
        handleFirstNameChange={handleFirstNameChange}
        handleLastNameChange={handleLastNameChange}
      />
    );
  }

  export default ParentComponent;
```

Then `Form` can become:

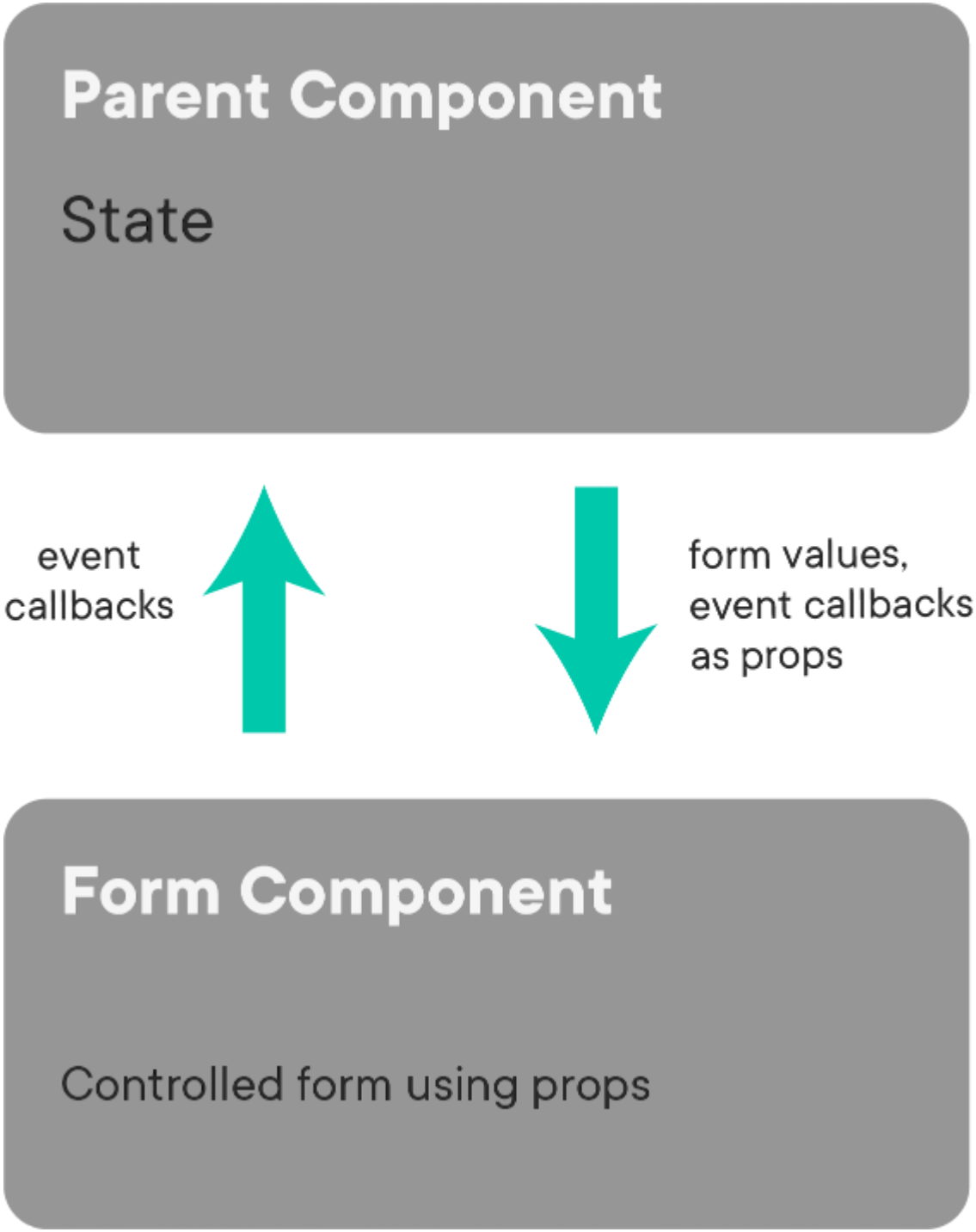
```
  // src/components/Form
  import React from "react";

  function Form({
    firstName,
    lastName,
    handleFirstNameChange,
    handleLastNameChange,
  }) {
    return (
      <form>
        <input type="text" onChange={handleFirstNameChange} value={firstName} />
        <input type="text" onChange={handleLastNameChange} value={lastName} />
        <button type="submit">Submit</button>
      </form>
    );
  }
```

```
export default Form;
```

Previously, our application was rendering `Form` directly inside `src/index.js`. Now, however, we've added a component that renders `Form` as a child. Because of this change, you'll need to update `src/index.js` so that it renders `ParentComponent` instead of `Form`.

> **Note**: If you're following along in the example files, don't forget to update `index.js` to point to `ParentComponent`. If you don't make this change, the behavior of the form inputs will appear the same, but they will just be regular HTML input fields — they will not be controlled. To verify this, you can log `event.target.value` from inside the `handleFirstNameChange` and `handleLastNameChange` functions in `ParentComponent`.

With `ParentComponent`, we've moved all the form logic up one level.


Parent Component
State
event callbacks
form values, event callbacks as props
Form Component
Controlled form using props

Being able to store controlled form data in other components opens some interesting doors for us. We could, for instance, create another component, a sibling of `Form`, that displays our form data as soon as a user starts filling in the form:

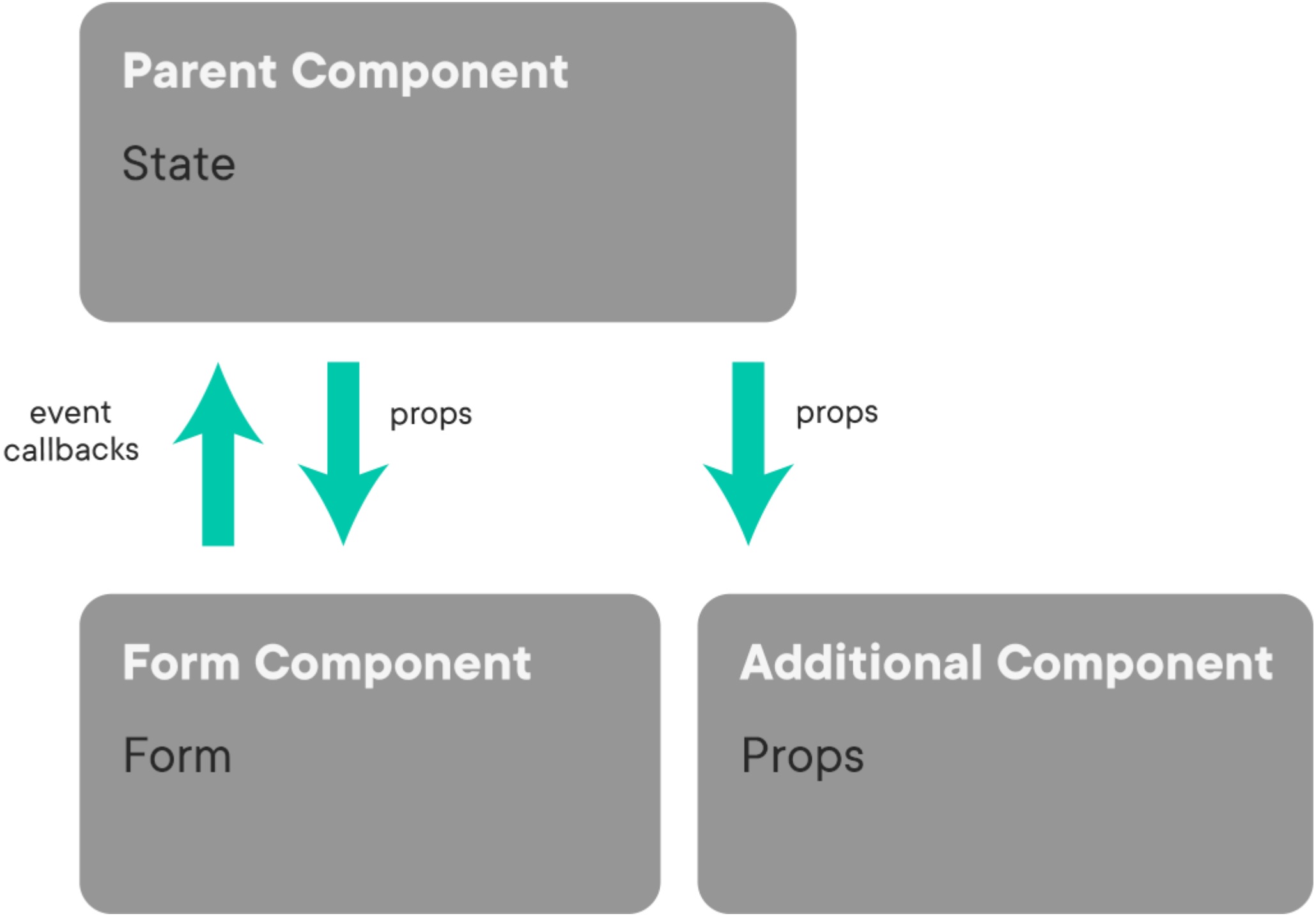
```
// src/components/DisplayData
import React from "react";

function DisplayData({ firstName, lastName }) {
  return (
    <div>
      <h1>{firstName}</h1>
      <h1>{lastName}</h1>
    </div>
  );
}

export default DisplayData;
```

...and adding it alongside `Form` (also wrapping both in a `div`):

```
// src/components/ParentComponent
import React, { useState } from "react";
import Form from "./Form";
import DisplayData from "./DisplayData";

function ParentComponent() {
  // ...
  return (
    <div>
      <Form
        firstName={firstName}
        lastName={lastName}
        handleFirstNameChange={handleFirstNameChange}
        handleLastNameChange={handleLastNameChange}
```

```
        />
        <DisplayData firstName={firstName} lastName={lastName} />
      </div>
    );
  }
```

Now we have a component that reads from the same state we're changing with the form.


Parent Component
State
event
callbacks
props
props
Form Component
Form
Additional Component
Props

This can be a very useful way to capture user input and utilize it throughout your application, even if a server is not involved.

The opposite can also be true. Imagine a user profile page with an 'Edit' button that opens a form for updating user info. When a user clicks that 'Edit' button, they expect to see a form with their user data pre-populated. This way, they can easily make small changes without rewriting all their profile info.

Just like we did with `ParentComponent`, this could be achieved by populating a form with data from props! After all, if we have a React app that is displaying user information, that information is stored *somewhere* on the app.

Controlled Forms for Validation

It might seem a little counterintuitive that we need to be so verbose when working with forms in React, but this actually opens the door to additional functionality. For example, let's say we want to write an input that only takes the numbers `0` through `5`. We can now validate the data the user enters *before* we set it on the state, allowing us to block any invalid values:

```
function Form() {
  const [number, setNumber] = useState(0);

  function handleNumberChange(event) {
    const newNumber = parseInt(event.target.value);
    if (newNumber >= 0 && newNumber <= 5) {
      setNumber(newNumber);
    }
  }

  return (
    <form>
      <input type="number" value={number} onChange={handleNumberChange} />
    </form>
  );
}
```

If the input is invalid, we simply avoid updating the state, preventing the input from updating. We could optionally set another state property (for example, `isInvalidNumber`). Using that state property, we can show an error in our component to indicate that the user tried to enter an invalid value:

```
function Form() {
  const [number, setNumber] = useState(0);
  const [isInvalidNumber, setIsInvalidNumber] = useState(null);

  function handleNumberChange(event) {
    const newNumber = parseInt(event.target.value);
    if (newNumber >= 0 && newNumber <= 5) {
      setNumber(newNumber);
      setIsInvalidNumber(null);
    } else {
      setIsInvalidNumber(`${newNumber} is not a valid number!`);
    }
  }

  return (
    <form>
      <input type="number" value={number} onChange={handleNumberChange} />
      {isInvalidNumber ? <span style={{ color: "red" }}>{isInvalidNumber}</span> : null}
    </form>
  );
}
```

If we tried to do this using an uncontrolled component, the input would be entered regardless, since we don't have control over the internal state of the input. In our `onChange` handler, we'd have to roll the input back to its previous value, which is pretty tedious!

Conclusion

Using a controlled component is the preferred way to do things in React — it allows us to keep *all* component state in the React state, instead of relying on the DOM to retrieve the element's value through its internal state.

Using a controlled form, whenever our state changes, the component re-renders, rendering the input with the new updated value. If we don't update the state, our input will not update when the user types. In other words, we need to update our input's state *programmatically*.

Resources

- **React Forms (https://reactjs.org/docs/forms.html)**