Controlled components vs. Uncontrolled components

This reading will teach you how to work with uncontrolled inputs in React and the advantages of controlled inputs via state design. You will also learn when to choose controlled or uncontrolled inputs and the features each option supports.

Introduction

In most cases, React recommends using controlled components to implement forms. While this approach aligns with the React declarative model, uncontrolled form fields are still a valid option and have their merit. Let's break them down to see the differences between the two approaches and when you should use each method.

Uncontrolled Inputs

Uncontrolled inputs are like standard HTML form inputs:

```
const Form = () => {
  return (
    <div>
      <input type="text" />
    </div>
  );
};
```

They remember exactly what you typed, being the DOM itself that maintains that internal state. How can you then get their value? The answer is by using a React ref.

In the code below, you can see how a ref is used to access the value of the input whenever the form is submitted.

```
const Form = () => {
  const inputRef = useRef(null);

  const handleSubmit = () => {
    const inputValue = inputRef.current.value;
    // Do something with the value
  }
  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef} type="text" />
    </form>
  );
};
```

In other words, you must **pull** the value from the field when needed.

Uncontrolled components are the simplest way to implement form inputs. There are certainly valued cases for them, especially when your form is straightforward. Unfortunately, they are not as powerful as their counterpart, so let's look at controlled inputs next.

Controlled Inputs

Controlled inputs accept their current value as a prop and a callback to change that value. That implies that the value of the input has to live in the React state somewhere. Typically, the component that renders the input (like a form component) saves that in its state:

```
const Form = () => {
  const [value, setValue] = useState("");

  const handleChange = (e) => {
    setValue(e.target.value)
  }

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

Every time you type a new character, the `handleChange` function is executed. It receives the new value of the input, and then it sets it in the state. In the code example above, the flow would be as follows:

- The input starts out with an empty string: `""`
- You type `"a"` and `handleChange` gets an `"a"` attached in the event object, as `e.target.value`, and subsequently calls `setValue` with it. The input is then updated to have the value of `"a"`.
- You type `"b"` and `handleChange` gets called with `e.target.value` being `"ab"`.and sets that to the state. That gets set into the state. The input is then re-rendered once more, now with `value = "ab"` .

This flow **pushes** the value changes to the form component instead of pulling like the ref example from the uncontrolled version. Therefore, the Form component always has the input's current value without needing to ask for it explicitly.

As a result, your data (React state) and UI (input tags) are always in sync. Another implication is that forms can respond to input changes immediately, for example, by:

- Instant validation per field
- Disabling the submit button unless all fields have valid data
- Enforcing a specific input format, like phone or credit card numbers

Sometimes you will find yourself not needing any of that. In that case uncontrolled could be a more straightforward choice.

The file input type

There are some specific form inputs that are always uncontrolled, like the file input tag.

In React, an `<input type="file" />` is always an uncontrolled component because its value is read-only and can't be set programmatically.

The following example illustrates how to create a ref to the DOM node to access any files selected in the form submit handler:

```
const Form = () => {
  const fileInput = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    const files = fileInput.current.files;
    // Do something with the files here
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        ref={fileInput}
        type="file"
      />
    </form>
  );
};
```

Conclusion

Uncontrolled components with refs are fine if your form is incredibly simple regarding UI feedback. However, controlled input fields are the way to go if you need more features in your forms.

Evaluate your specific situation and pick the option that works best for you.

The below table summarizes the features that each one supports:

Feature	Uncontrolled	Controlled
One-time value retrieval (e.g. on submit)	**Yes**	**Yes**
Validating on submit	**Yes**	**Yes**
Instant field validation	No	**Yes**
Conditionally disabling a submit button	No	**Yes**
Enforcing a specific input format	No	**Yes**
Several inputs for one piece of data	No	**Yes**
Dynamic inputs	No	**Yes**

And that's it about controlled vs. uncontrolled components. You have learned in detail about each option, when to pick one or another, and finally, a comparison of the features supported.