Controlled and Uncontrolled Inputs

How inputs can be handled in React.

WE'LL COVER THE FOLLOWING

The two terms *controlled* and *uncontrolled* are very often used in the context of forms management.

Controlled Inputs

A *controlled* input is an input that gets its value from a single source of truth. For example, the `App` component below has a single `<input>` field which is *controlled*:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
  }
  render() {
    return <input type='text' value={ this.state.value } />;
  }
};
```

The result of this code is an input element that we can focus, but can't change. It is never updated because we have a single source of truth - the `App`'s component state.

To make the input work as expected we have to add an `onChange` handler and update the state (the single source of truth), which will trigger a new rendering cycle and we will see what we typed.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
    this._change = this._handleInputChange.bind(this);
  }
  render() {
    return (
      <input
        type='text'
        value={ this.state.value }
        onChange={ this._change } />
    );
  }
  _handleInputChange(e) {
    this.setState({ value: e.target.value });
  }
};
```

Uncontrolled Input

On the other hand, we have *uncontrolled* input, where we let the browser handle the user's updates. We may still provide an initial value by using the `defaultValue` prop but after that, the browser is responsible for keeping the state of the input.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
  }
  render() {
    return <input type='text' defaultValue={ this.state.value } />
  }
};
```

That `<input>` element above is a little bit useless because the user updates the value but our component has no idea about that. We then have to use `Refs` to get access to the actual element.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'hello' };
    this._change = this._handleInputChange.bind(this);
  }
  render() {
    return (
      <input
```

```
          type='text'
          defaultValue={ this.state.value }
          onChange={ this._change }
          ref={ input => this.input = input }/>
      );
    }
    _handleInputChange() {
      this.setState({ value: this.input.value });
    }
  };
```

The `ref` prop receives a string or a callback. The code above uses a callback and stores the DOM element into a *local* variable called `input`. Later when the `onChange` handler is fired we get the new value and send it to the `App`'s state.

Using a lot of `refs` is not a good idea. If it happens in your app consider using `controlled` inputs and re-think your components.

Final thoughts

The choice between *controlled* and *uncontrolled* inputs is very often underrated. However, I believe that it is a fundamental decision because it dictates the data flow in the React component. Given below is code for how uncontrolled inputs would be handled. However, I personally think that *uncontrolled* inputs are kind of an anti-pattern and would avoid them when possible.

```
import React from 'react';
require('./style.css');

import ReactDOM from 'react-dom';
import App from './app.js';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Now that we have discussed the different types of inputs React components can take, we will proceed to look into the different types of components one can have created in React in the following section.

Quick Quiz on Controlled and Uncontrolled Inputs!

Q

When we update our value from a single source of truth, it is an example of uncontrolled input.

COMPLETED 0%

1 of 1