Using hooks

Now that you understand what hooks are in React and have some basic knowledge on the `useState` hook, let's dive in deeper. In this reading, you will learn how to use hooks in React components and understand the use-cases for the `useState` hook.

Let's say you have a component with an input text field. The user can type into this text field. The component needs to keep track of what the user types within this text field. You can add state and use the `useState` hook, to hold the string.

As the user keeps typing, the local state that holds the string needs to get updated with the latest text that has been typed.

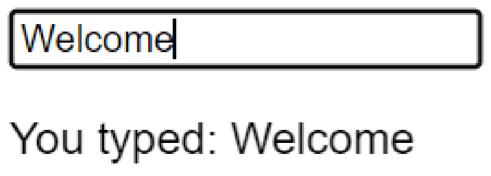
Let's discuss the below example.

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

export default function InputComponent() {
  const [inputText, setText] = useState('hello');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <>
      <input value={inputText} onChange={handleChange} />
      <p>You typed: {inputText}</p>
      <button onClick={() => setText('hello')}>
        Reset
      </button>
    </>
  );
}
```

To do this, let's define a React component and call it `InputComponent`. This component renders three things:

- An input text field
- Any text that has been entered into the field
- A Reset button to set the field back to its default state

As the user starts typing within the text field, the current text that was typed is also displayed.


Welcome

You typed: Welcome

The state variable **`inputText`** and the **`setText`** method are used to set the current text that is typed. The **`useState`** hook is initialized at the beginning of the component.

```javascript
const[inputText, setText] = useState('hello');
```

By default, the **`inputText`** will be set to “hello”.

As the user types, the **`handleChange`** function, reads the latest input value from the browser’s input DOM element, and calls the **`setText`** function, to update the local state of **`inputText`**.

```javascript
function handleChange(e) {
    setText(e.target.value);
};
```

Finally, clicking the reset button will update the **`inputText`** back to “hello”.

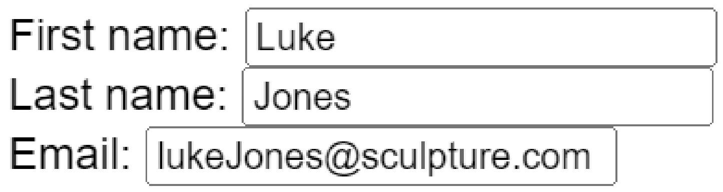
Isn’t this neat?

Keep in mind that the **`inputText`** here is local state and is local to the **`InputComponent`**. This means that outside of this component, **`inputText`** is unavailable and unknown. In React, state is always referred to the local state of a component.

Hooks also come with a set of rules, that you need to follow while using them. This applies to all React hooks, including the **`useState`** hook that you just learned.

- You can only call hooks at the top level of your component or your own hooks.
- You cannot call hooks inside loops or conditions.
- You can only call hooks from React functions, and not regular JavaScript functions.

To demonstrate, let’s extend the previous example, to include three input text fields within a single component. This could be a registration form with fields for first name, last name and email.



First name: Luke
Last name: Jones
Email: lukeJones@sculpture.com

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

export default function RegisterForm() {
  const [form, setForm] = useState({
    firstName: 'Luke',
    lastName: 'Jones',
    email: 'lukeJones@sculpture.com',
  });

  return (
    <>
      <label>
        First name:
        <input
          value={form.firstName}
          onChange={e => {
            setForm({
              ...form,
              firstName: e.target.value
            });
          }}
        />
      </label>
      <label>
        Last name:
        <input
          value={form.lastName}
          onChange={e => {
            setForm({
              ...form,
              lastName: e.target.value
            });
          }}
        />
      </label>
      <label>
        Email:
        <input
          value={form.email}
          onChange={e => {
```

Notice that you are using a `form` object to store the state of all three text input field values:

```
const[form, setForm] =useState({
firstName:'Luke',
lastName:'Jones',
    email:'lukeJones@sculpture.com',
});
```

You do not need to have three separate state variables in this case, and instead you can consolidate them all together into one **form** object for better readability.

In addition to the **useState** hook, there are other hooks that come in handy such as **useContext**, **useMemo**, **useRef**, etc. When you need to share logic and reuse the same logic across several components, you can extract the logic into a custom hook. Custom hooks offer flexibility and can be used for a wide range of use-cases such as form handling, animation, timers, and many more.

Next, I'll give you an explanation of how the useRef hook works.

The useRef hook

We use the **useRef** hook to access a child element directly.

When you invoke the **useRef** hook, it will return a **ref** object. The **ref** object has a property named **current**.

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}

```

Using the ref attribute on the input element, I can then access the current value and invoke the focus() method on it, thereby focusing the input field.

There are situations where accessing the DOM directly is needed, and this is where the useRef hook comes into play.

Conclusion

In this reading, you have explored hooks in detail and understand how to use the **useState** hook to maintain state within a component. You also understand the benefits of using hooks within a React component.