Mastering Hooks and Functional Components

CS571 – Mobile Application Development

Maharishi University of Management

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

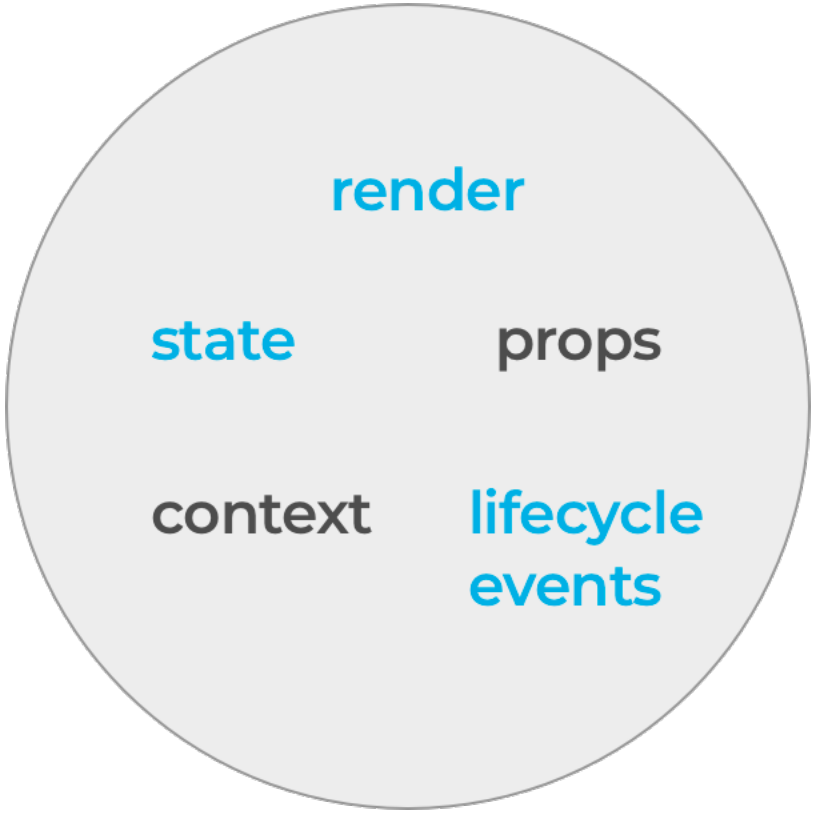
All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

Component API

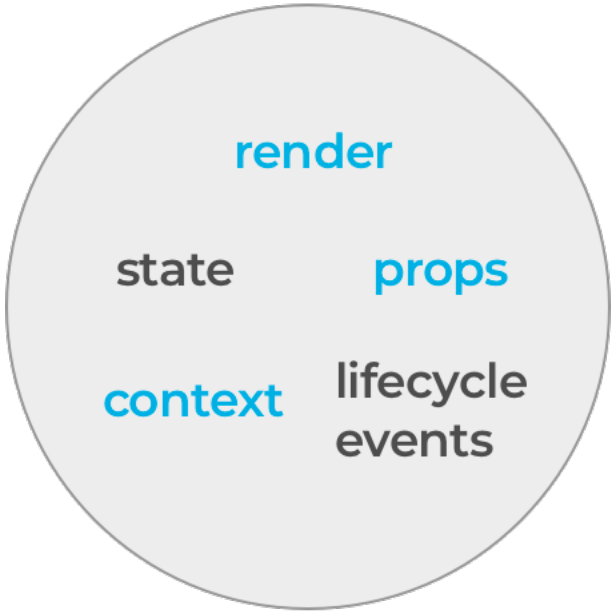
There are five API's available to every component, and they are:

- `render`
- `state`
- `props`
- `context`
- `lifecycle events`

Stateful vs. Stateless



Stateful



Stateless

Stateless to Stateful Hooks

Refer to React documentation and read about the following hooks:

- `useState()`
- `useEffect()`
- `useMemo()`
- `useReducer()`
- `useContext()`
- `useRef()`
- `Custom Hooks`

Reading and Updating State

To track state updates and trigger virtual DOM diffing and real DOM reconciliation, React needs to be aware of any changes that happen to any state property that are used within components.

This is where the **`useState`** hook comes into play. It defines a state element and give us back a getter and setter for it.

React.useState()

The **`useState`** function returns an array with exactly 2 items.

- The first item is a **value** (getter)
- The second item is a **function** (setter).

We like to use array destructuring to give these items names.

```
const [count, setCount] = React.useState(0);
```

useState() Example

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

const Button = () => {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      {count}
    </button>
  );
};

ReactDOM.render(<Button />, mountNode);
```

What happens when React re-render the component?

this.setState() vs useState()

```
export default class App extends React.Component{
   state = {
      first: 'Asaad',
      last: 'Saad'
   }
   changeFirstName = ()=> this.setState({first: 'Mike'});

   render(){
      return (
          <div>
              <button onClick={this.changeFirstName}>Change Name<button>
              <p>{this.state.first} {this.state.last}</p>
          </div>
      );
   }
}
```

this.setState() vs useState()

```
export default function App() {
   const [state, setState] = React.useState({ first: 'Asaad', last: 'Saad'});

   const changeFirstName = ()=> setState({first: 'Mike'});

   return (
      <div>
         <button onClick={changeFirstName}>Change Name<button>
         <p>{state.first} {state.last}</p>
      </div>
   );
}
```

this.setState() vs useState() - Fix

```
export default function App() {
   const [state, setState] = React.useState({ first: 'Asaad', last: 'Saad'});

   const changeFirstName = ()=> setState({...state, first: 'Mike'});

   return (
      <div
         <button onClick={changeFirstName}>Change Name<button>
         <p>{state.first} {state.last}</p>
      </div>
   );
}
```

useRef()

```
function App() {
    const header = React.useRef();

    const change = () => { header.current.innerHTML = "Bonjour"};

    return (
        <div>
            <h1 ref={header}>Hello</h1>
            <button onClick={change}>French?</button>
        </div>
    );
}
```

What is the difference between **`useRef()`** used for function components and **`createRef()`** for class components?

useEffect()

For function components, side effects are managed using the **`React.useEffect`** hook function.

It removes the need for **`componentDidMount`** , **`componentDidUpdate`** and **`componentWillUnmount`** and **`componentShouldUpdate`** because it handles the use case of all these life cycle methods.

It takes 2 arguments:

- Callback function
- Array of dependencies

useEffect() Example

If we pass a function:

```
React.useEffect(() => {
    // Called after every render
});
```

This will cause **`useEffect`** to run after every render, just like **`componentDidUpdate`**

Do not use it a lot because you don't want the side effects to run after every render.

Return from useEffect()

When you return a function from `useEffect()`, the function will be called before the next render

```
React.useEffect(() => {
   console.log(`Effect`);
   // cleanup before next render
   return () => console.log(`Clean up`);
 });
```

Dependencies

The first time React renders a component that has a **`useEffect`** call it'll invoke its callback function. After each new render of that component, only if the values of the dependencies are different from what they used to be in the previous render, React will invoke the callback function again.

```
useEffect(() => {}, [dep1, dep2]);
```

Dependencies Example

```
React.useEffect(() => {
   // First render
   // Next render only if user_id is changed
   console.log(`Effect dep`);
 }, [user_id]);
```

Example

```
function App() {
  const [count1, setCount1] = React.useState(0);
  const [count2, setCount2] = React.useState(0);
  React.useEffect(() => {
     console.log(`Effect Start`);
     return () => console.log(`Effect End`);
  }, [count2]);
  return (
   <div>
      {count1}, {count2}
      <button onClick={ _=> setCount1(count1 + 1)}>inc count1</button>
      <button onClick={ _=> setCount2(count2 + 1)}>inc count2</button>
   </div>
  );
}
```

Empty Dependencies

When we pass an empty array of dependencies, we tell React that **`useEffect`** should never be called on every render, this is similar to using **`componentDidMount`** and **`componentWillUnmount`**

```
React.useEffect(() => {
    // Mount
    console.log(`Effect empty deps`);
    // Unmount
    return () => console.log(`Unmount`);
 }, []);
```

useMemo()

useMemo will only recompute the memoized value when one of the deps have changed. It returns a memoized value.

```
function App() {
  const [counter1, setCounter1] = React.useState(0);
  const [counter2, setCounter2] = React.useState(0);
  const compute = () => { console.log(`Intensive work`); return `Computed Results` };
  const computed = React.useMemo(() => compute(), [counter1]);
  return (
    <div>
      <h1>{counter1}, {counter2}</h1>
      <button onClick={() => setCounter1(counter1 + 1)}>INC1</button>
      <button onClick={() => setCounter2(counter2 + 1)}>INC2</button>
      <h2>{computed}</h2>
    </div>
  );
}
```

Custom Hooks

A custom Hook is a function whose name starts with **`"use"`** and that may call other Hooks.

A custom Hook doesn't need to have a specific signature, **we can decide what it takes as arguments, and what it should return**. It's just a normal function.

Custom Hook Example

```
const useFetch = function(id) {
   const [state, setState] = useState({ data: null, loading: false });
   useEffect(() => {
      setState(prevState => ({ ...prevState, loading: true }));
      // Fetch Data by id, then:
      setState(prevState => ({...prevState, data: `Data ${id}`, loading: false}));
   }, [id]);

   return state;
};
```

What happens when we don't pass **id** as dependency?
What does this hook return?

Using our custom Hook

```
function App() {
   const [id, setId] = useState(1);
   const { data, loading } = useFetch(id);

   return (
      <div>
         <button onClick={() => setId(id + 1)}>{id}</button>
         <h1>{loading ? "loading..." : data}</h1>
      </div>
   );
}
```

useReducer()

Use it when your state is complex, otherwise, use **`useState()`**

```
const reducer = function(state, action) {
   switch (action.type) {
      case "SET_NAME": return { ...state, name: action.payload };
      case "SET_GRADE": return { ...state, grade: action.payload };
      default: return state;
   }
};
const [{ name, grade }, dispatch] = React.useReducer(reducer, { name: '', grade: 0 });

dispatch({ type: "SET_NAME", payload: "Asaad Saad" })
dispatch({ type: "SET_GRADE", payload: 100 })
```

React Context

To create a global Context object we call **`React.createContext()`**, this object can be provided and consumed with any value, ideally the value is your "state". You may optionally pass a default value.

```
MyCountContext = React.createContext();
```

Provide the Context

You can provide context to your App using the value attribute:

```
const [count, setCount] = React.useState(0);

<MyCountContext.Provider value={{count, setCount}}>
      // App
</MyCountContext.Provider>
```

Consume the Context

To consume the context from any child component:

```
<MyCountContext.Consumer>
      {({ count, setCount }) => <div>Count Value: {count}</div>}
</MyCountContext.Consumer>
 // OR
const { count, setCount } = React.useContext(MyCountContext);
```

Performance

Sometimes we want a component to only renders when its props are changed.

- Extend your class component from **`React.PureComponent`**
- Wrap your function component with **`React.memo()`**

`PureComponent` is exactly the same as **`Component`** except that it handles the **`shouldComponentUpdate`** method for you. It re-renders only when **`props`** or **`state`** changes, **`PureComponent`** will do a shallow comparison on both **`props`** and **`state`**.

React.memo()

`React.memo()` is the functional way of creating **`PureComponent`** in class-based components.

Components will only rerender if its props have changed! Normally all React components starting of the component who marked as dirty will go through a render when changes are made. With **`PureComponent`** and **`React.memo()`**, we can have only some components render.

This is a performance boost since only the things that need to be rendered are rendered.