React CheatSheet

CODE HELP

In this Cheatsheet, we will cover the basics of React. We will provide examples to help you understand how React works and how to use them in your own web development projects. Whether you are a beginner or an experienced developer, this PDF can serve as a useful reference guide.

REACT

React is a JavaScript library for building user interfaces. It was developed and is maintained by Facebook and a community of individual developers and companies. React allows developers to build reusable UI components, manage the state of their applications, and render dynamic updates efficiently.

React uses a virtual DOM (Document Object Model) to update the view without having to reload the entire page. This results in fast and smooth updates to the user interface. React also uses a component-based architecture, which allows developers to easily reuse and compose their components to build complex UIs.

React is used for both web and mobile development, and is popular for its simplicity, versatility, and performance. Many companies and organizations use React for their websites and applications, including Facebook, Airbnb, Netflix, and Dropbox.

MANAGE STATE

useState

```
const [count, setCount] = useState(initialCount);
```

- Class way

```
const initialCount = 0;
class Counter extends Component {
      constructor(props) {
            super(props);
            this.state = { count : initialCount };
      }
      render() {
         return (
         <div>
            <p>You clicked {this.state.count} times</p>
            <button
            onClick = {() => this.setState(({count}) => ({ count: count + 1 }))}
            >
            Click me
            </button>
         </div>
      );
   }
}
```

- Hook Way

```
import { useState } from "react";
const initialCount = 0;
function Counter() {
	const [count, setCount] = useState(initialCount);
	return (
	<div>
		<p>You clicked {count} times</p>
		<button
		onClick = {()  =>  setCount((c) => c + 1)}
		>
			Click me
		</button>
	</div>
   );
}
```

useReducer

```
const [state, dispatch] = useReducer(
	reducer,
	initialState,
	initialDispatch
);
```

An alternative to useState. Use it in the components that need complex state management, such as multiple state values being updated by multiple methods.

```
const initialState = {count: 0};
function reducer(state, action) {
      switch (action.type) {
            case 'increment':
                  return {count: state.count + 1};
            case 'decrement':
                  return {count: state.count - 1};
            default:
                  throw new Error();
            }
      }
function Counter({initialState}) {
      const [state, dispatch] = useReducer(reducer, initialState);
      return (
      <>
      Count: {state.count}
      <button onClick = {() => dispatch({type: 'increment'})}>+</button>
      <button onClick = {() => dispatch({type: 'decrement'})}>+</button>
      </>
   );
}
```

HANDLE SIDE EFFECTS

useEffect

```
useEffect(() => {
      applyEffect(dependencies);
      return () => cleanupEffect();
}, [dependencies]);
```

- Class way

```
class FriendStatus extends React.Component {
	state = { isOnline : null };
	componentDidMount() {
		ChatAPI.subscribeToFriendStatus(
		this.props.friend.id,
		this.handleStatusChange
		);
	}
	componentWillUnmount() {
		ChatAPI.unsubscribeFromFriendStatus(
		this.props.friend.id,
		this.handleStatusChange
		);
	}
	componentDidUpdate(prevProps) {
		if(this.props.friend.id !== prevProps.id) {
			ChatAPI.unsubscribeFromFriendStatus(
			prevProps.friend.id,
			this.handleStatusChange
			);
			ChatAPI.subscribeToFriendStatus(
			this.props.friend.id,
			this.handleStatusChange
			);
		}
	}
	handleStatusChange = status => {
		this.setState({
		isOnline: status.isOnline
	});
 }
render() {
	if (this.state.isOnline === null) {
		return 'Loading…';
	}
	return this.state.isOnline ? 'Online' : 'Offline';
	}
}
```

- Hook Way

```
import { useState, useEffect } from 'react';
function FriendStatus(props) {
        const [isOnline, setIsOnline] = useState(null);
        status
        const handleStatusChange = (status) => setIsOnline(status.isOnline);
        useEffect(
        () => {
 // Update status when the listener triggers
        ChatAPI.subscribeToFriendStatus(
        props.friend.id,
        handleStatusChange
        );
 // Stop listening to status changes every time we cleanup
         return function cleanup() {
                ChatAPI.unsubscribeFromFriendStatus(
                props.friend.id,
                handleStatusChange
                );
        };
 },
 [props.friend.id] // Cleanup when friend id changes or when we "unmount"
 );
if (isOnline === null) {
        return 'Loading...';
 }
return isOnline ? 'Online' : 'Offline';
}
```

useLayoutEffect

```
useLayoutEffect(() => {
        applyBlockingEffect(dependencies);
        return cleanupEffect();
}, [dependencies]);
```

useLayoutEffect is almost same as useEffect, but fires synchronously after the render phase. Use this to safely read from or write to the DOM

```
import { useRef, useLayoutEffect } from "react";
function ColoredComponent({color}) {
	const ref = useRef();
	useLayoutEffect(() => {
		const refColor = ref.current.style.color;
		console.log(`${refColor} will always be the same as ${color}`);
		ref.current.style.color = "rgba(255,0,0)";
	}, [color]);
	useEffect(() => {
		const refColor = ref.current.style.color;
		console.log(
		`but ${refColor} can be different from ${color} if you play with the
DOM`
		);
	}, [color]);
	return (
		<div ref={ref} style={{ color: color }}>
			Hello React hooks !
		</div>
	);
}
```

```
<ColoredComponent color = {"rgb(42, 13, 37)"}/>
// rgb(42, 13, 37) will always be the same as rgb(42, 13, 37)
// but rgb(255, 0, 0) can be different from rgb(42, 13, 37) if you play with the DOM
```

USE THE CONTEXT API

useContext

```
const ThemeContext = React.createContext();
const contextValue = useContext(ThemeContext);
```

- Class way

```
class Header extends React.Component {
        public render() {
        return (
        <AuthContext.Consumer>
        {({ handleLogin, isLoggedIn}) => (
        <ModalContext.Consumer>
        {({ isOpen, showModal, hideModal }) => (
        <NotificationContext.Consumer>
        {({ notification, notify }) => {
                return (
                ...
                )
        }}
        </NotificationContext.Consumer>
        )}
        </ModalContext.Consumer>
        )}
        </AuthContext.Consumer>
        );
    }
}
```

- Hook Way

```
import { useContext } from 'react';
function Header() {
	const { handleLogin, isLoggedIn} = useContext(AuthContext);
	const { isOpen, showModal, hideModal } = useContext(ModalContext);
	const { notification, notify } = useContext(NotificationContext);

	return ...
}
```

NOTE: Use the created context, not the consumer.

```
const Context = React.createContext(defaultValue);
// Wrong
const value = useContext(Context.Consumer);
// Right
const value = useContext(Context);
```

MEMOIZE EVERYTHING

useMemo

```
const  memoizedValue  =  useMemo (
	()  =>  expensiveFn (dependencies),
	[dependencies]
);
```

```
 function RandomColoredLetter(props) {
        const [color, setColor] = useState('#fff')
        const [letter, setLetter] = useState('a')
        const handleColorChange = useMemo(() => () =>
setColor(randomColor()), []);
        const handleLetterChange = useMemo(() => () =>
setLetter(randomColor()), []);
        return (
    <div>
        <ColorPicker handleChange={handleColorChange} color={color} />
        <LetterPicker handleChange={handleLetterChange} letter={letter} />
        <hr/>
        <h1 style={{color}}>{letter}</h1>
    </div>
  )
 }
```

useCallback

```
 const  memoizedCallback  =  useCallback (
        expensiveFn (dependencies),
        [dependencies]
 );
```

```
 function RandomColoredLetter(props) {
   const [color, setColor] = useState('#fff')
   const [letter, setLetter] = useState('a')
   const handleColorChange = useCallback(() => setColor(randomColor()), []);
   const handleLetterChange = useCallback(() => setLetter(randomColor()), []);
   return (
   <div>
       <ColorPicker handleChange={handleColorChange} color={color} />
       <LetterPicker handleChange={handleLetterChange} letter={letter} />
       <hr/>
       <h1 style={{color}}>{letter}</h1>
   </div>
  )
 }
```

USE REFS

useRef

```
const ref = useRef();
```

useRef can just be used as a common React ref :

```
import { useRef } from "react";
function TextInput() {
      const inputRef = useRef(null);
      const onBtnClick = () => inputRef.current.focus();
      return (
      <>
      <input ref={ref} />
      <button onClick={onBtnClick}>Focus the text input</button>
      </>
   )
}
```

But it also allows you to just hold a mutable value through any render. Also, mutating the value of ref.current will not cause any render.

useImperativeHandle

```
useImperativeHandle (
      ref,
      createHandle,
      [dependencies]
)
```

useImperativeHandle allows you to customize the exposed interface of a component when using a ref. The following component will automatically focus the child input when mounted :

```
function TextInput(props, ref) {
	const inputRef = useRef(null);
	const onBtnClick = () => inputRef.current.focus();
	useImperativeHandle(ref, () => ({
		focusInput: () => inputRef.current.focus();
	});
	return (
	<Fragment>
	<input ref={inputRef} />
	<button onClick={onBtnClick}>Focus the text input</button>
	</Fragment>
	)
}
const TextInputWithRef = React.forwardRef(TextInput);
function Parent() {
	const ref = useRef(null);
	useEffect(() => {
		ref.focusInput();
	}, []);
	return (
	<div>
	<TextInputWithRef ref={ref} />
	</div>
	);
}
```

REUSABILITY

Extract reusable behaviour into custom hooks.

```
import { useState, useRef, useCallback, useEffect } from "React";
// let's hide the complexity of listening to hover changes
function useHover() {
	const [value, setValue] = useState(false); // store the hovered state
	const ref = useRef(null); // expose a ref to listen to
 // memoize function calls
	const handleMouseOver = useCallback(() => setValue(true), []);
	const handleMouseOut = useCallback(() => setValue(false), []);
 // add listeners inside an effect,
 // and listen for ref changes to apply the effect again
	useEffect(() => {
		const node = ref.current;
		if (node) {
			node.addEventListener("mouseover", handleMouseOver);
			node.addEventListener("mouseout", handleMouseOut);
		return () => {
			node.removeEventListener("mouseover", handleMouseOver);
			node.removeEventListener("mouseout", handleMouseOut);
			};
		}
	}, [ref.current]);
 // return the pair of the exposed ref and it's hovered state
	return [ref, value];
}
```

```
const HoverableComponent = () => {
	const [ref, isHovered] = useHover();
	return (
	<span style={{ color: isHovered ? "blue" : "red" }} ref={ref}>
		Hello React hooks !
	</span>
	);
};
```