React Cheatsheet

A comprehensive cheat sheet covering React fundamentals, JSX syntax, component lifecycle, hooks, and common patterns, designed to provide a quick reference for React developers.

React Fundamentals

JSX Syntax

JSX (JavaScript XML): Allows writing HTML-like syntax in JavaScript.

Example:

```
const element = <h1>Hello, React!</h1>;
```

Embedding Expressions: Use curly braces `{}` to embed JavaScript expressions.

Example:

```
const name = 'User';
const element = <h1>Hello, {name}!</h1>;
```

JSX Attributes: Define HTML attributes using JSX syntax.

Example:

```
const element = <img src={user.imageUrl} alt="{user.name}" />;
```

Conditional Rendering: Use ternary operators or `&&` operator for conditional rendering.

Example:

```
{isLoggedIn ? <LogoutButton /> : <LoginButton />}
```

Rendering Lists: Use `.map()` to render lists of elements.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

Fragments: Use `<> </>` or `<React.Fragment>` to group multiple elements without adding an extra node to the DOM.

Example:

```
<><h1>Title</h1><p>Description</p></>
```

Components

Function Components: Simple components that accept props and return JSX.

Example:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Class Components: Components that use ES6 classes, extend `React.Component`, and have a `render()` method.

Example:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Props: Data passed from parent to child components. Props are immutable from the component's perspective.

Example:

```
<Welcome name="Sara" />
```

State: Internal data of a component that can change over time. Changes to state trigger re-rendering.

Example:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <h1>Count: {this.state.count}</h1>;
  }
}
```

Handling Events: React events are named using camelCase and pass a function as the event handler.

Example:

```
<button onClick={this.handleClick}>Click me</button>
```

Component Composition: Building complex UIs by composing simpler components.

Example:

```
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
    </div>
  );
}
```

Lifecycle Methods & Hooks

Lifecycle Methods (Class Components)

`constructor(props)`: Initializing state and binding event handlers.

Example:

```
constructor(props) {
  super(props);
  this.state = { count: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

`render()`: Required method for class components, returns JSX.

Example:

```
render() {
  return <h1>Count: {this.state.count}</h1>;
}
```

`componentDidMount()`: Invoked immediately after a component is mounted (inserted into the tree).

Example:

```
componentDidMount() {
  document.title = `You clicked ${this.state.count} times`;
}
```

`componentDidUpdate(prevProps, prevState)`: Invoked immediately after updating occurs.

Example:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

`componentWillUnmount()`: Invoked immediately before a component is unmounted and destroyed.

Example:

```
componentWillUnmount() {
  clearInterval(this.intervalId);
}
```

Hooks (Function Components)

`useState()`: Enables function components to have state.

Example:

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

function Example() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

`useEffect()`: Performs side effects in function components.

Example:

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

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]); // Only re-run the effect if count changes

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

`useContext()`: Accepts a context object and returns the current context value.

Example:

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

const ThemeContext = React.createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button theme={theme}>I am styled by theme context!</button>;
}
```

`useReducer()`: Manages complex state logic.

Example:

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

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() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </>
  );
}
```

`useCallback()`: Memoizes a callback function.

Example:

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

function Example() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={increment}>Click me</button>
    </div>
  );
}
```

`useMemo()`: Memoizes a value.

Example:

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

function Example({ a, b }) {
  const result = useMemo(() => {
    return a + b;
  }, [a, b]);

  return <div>Result: {result}</div>;
}
```

Advanced Concepts

Context API

Creating a Context:

```
const MyContext =
React.createContext(defaultValue);
```

Providing a Context:

```
<MyContext.Provider value={/* some value */}>
  {/* children */}
</MyContext.Provider>
```

Consuming a Context:

```
<MyContext.Consumer>
  {value => /* render something based on the
context value */}
</MyContext.Consumer>
```

Or using `useContext` hook:

```
const value = useContext(MyContext);
```

Higher-Order Components (HOCs)

Definition: A function that takes a component and returns a new, enhanced component.

Example:

```
function withLogging(WrappedComponent) {
  return class WithLogging extends
React.Component {
    componentDidMount() {
      console.log('Component mounted:',
WrappedComponent.name);
    }

    render() {
      return <WrappedComponent {...this.props}
/>;
    }
  };
}

const EnhancedComponent =
withLogging(MyComponent);
```

Render Props

Definition: A technique for sharing code between React components using a prop whose value is a function.

Example:

```
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove = (event) => {
    this.setState({ x: event.clientX, y:
event.clientY });
  }

  render() {
    return (
      <div style={{ height: '100vh' }}
onMouseMove={this.handleMouseMove}>
        {this.props.render(this.state)}
      </div>
    );
  }
}

function App() {
  return (
    <Mouse render={mouse => (
      <p>The mouse position is ({mouse.x},
{mouse.y})</p>
    )}/>
  );
}
```

Error Boundaries

Definition: React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI.

Example:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will
show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error
reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Common Patterns and Best Practices

Controlled vs. Uncontrolled Components

Controlled Component: Form data is handled by the React component's state. The component controls the input's value.

Example:

```
function ControlledInput() {
  const [value, setValue] = useState('');

  const handleChange = (event) => {

setValue(event.target.value);
  };

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

Uncontrolled Component: Form data is handled by the DOM itself. Use `ref` to access the input's value.

Example:

```
function UncontrolledInput() {
  const inputRef = useRef(null);

  const handleSubmit = (event) => {

event.preventDefault();
    alert(`Value: ${inputRef.current.value}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Code Splitting

Definition: Breaking down the application into smaller chunks, loading only the necessary code for a particular route or feature. Achieved using `React.lazy` and `Suspense`.

Example:

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

const MyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <MyComponent />
    </Suspense>
  );
}
```

Memoization

Definition: Optimizing performance by preventing unnecessary re-renders. Use `React.memo` for functional components and `shouldComponentUpdate` for class components.

Example:

```
const MyComponent = React.memo(function MyComponent(props) {
  /* only re-renders if props change */
  return <div>{props.value}</div>;
});
```