Cheat Sheet for comprensive React

Comprehensive React Cheat Sheet

1. Introduction to React

- **React** is a JavaScript library for building user interfaces.

- **Declarative**: React makes it painless to create interactive UIs.

- **Component-Based**: Build encapsulated components that manage their own state.

- **Learn Once, Write Anywhere**: You can develop new features without rewriting existing code.

---

2. Basic Concepts

Components

- **Functional Components**: Simple JavaScript functions that return JSX.

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

- **Class Components**: ES6 classes that extend `React.Component`.

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

JSX

- **JSX** is a syntax extension to JavaScript, similar to a template language.

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

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

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

Props

- **Props** are read-only properties passed to a component.

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

- **Default Props**: Set default values for props.

```
Welcome.defaultProps = {
  name: 'Stranger'
};
```

State

- **State** is an object that holds data that may change over time.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

- **Updating State**: Use `setState()` to update state.

```
this.setState({ date: new Date() });
```

___

3. Component Lifecycle

Mounting

- **constructor()**: Initialize state and bind methods.
- **render()**: Return JSX.
- **componentDidMount()**: Called after the component is mounted.

Updating

- **render()**: Re-render when props or state change.
- **componentDidUpdate()**: Called after the component updates.

Unmounting

- **componentWillUnmount()**: Cleanup before the component is removed.

---

4. Hooks

useState

- **useState**: Manage state in functional components.

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

useEffect

- **useEffect**: Perform side effects in functional components.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);
```

useContext

- **useContext**: Access context in functional components.

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

useReducer

- **useReducer**: Manage complex state logic.

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

useCallback

- **useCallback**: Memoize functions to prevent unnecessary re-renders.

```
  const memoizedCallback = useCallback(() => {
    doSomething(a, b);
  }, [a, b]);
```

useMemo

- **useMemo**: Memoize values to prevent unnecessary calculations.

```
  const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a,
b]);
```

useRef

- **useRef**: Access DOM elements or persist values across renders.

```
  const inputRef = useRef(null);
```

____

5. Advanced Topics

Context API

- **Context**: Share data across the component tree without passing props.

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

Higher-Order Components (HOC)

- **HOC**: A function that takes a component and returns a new component.

```
  const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Render Props

- **Render Props**: A technique for sharing code between React components using a prop whose value is a function.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>
```

Error Boundaries

- **Error Boundaries**: Catch JavaScript errors anywhere in their child component tree.

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

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Portals

- **Portals**: Render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container);
```

Refs

- **Refs**: Access and manipulate DOM elements directly.

```
const inputRef = useRef(null);
```

____

6. Performance Optimization

React.memo

- **React.memo**: Memoize functional components to prevent unnecessary re-renders.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

useCallback

- **useCallback**: Memoize functions to prevent unnecessary re-renders.

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

useMemo

- **useMemo**: Memoize values to prevent unnecessary calculations.

```
  const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a,
b]);
```

Lazy Loading

- **Lazy Loading**: Load components only when they are needed.

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

Code Splitting

- **Code Splitting**: Split your code into smaller bundles that can be loaded on demand.

```
import('./math').then(math => {
  console.log(math.add(16, 26));
});
```

____

7. Forms and Events

Controlled Components

- **Controlled Components**: Form elements whose values are controlled by React state.

```
  class NameForm extends React.Component {
    constructor(props) {
      super(props);
      this.state = { value: '' };
    }

    handleChange = (event) => {
      this.setState({ value: event.target.value });
    }

    handleSubmit = (event) => {
      alert('A name was submitted: ' + this.state.value);
      event.preventDefault();
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Name:
            <input type="text" value={this.state.value}
onChange={this.handleChange} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

Uncontrolled Components

- **Uncontrolled Components**: Form elements that maintain their own state.

```
  class NameForm extends React.Component {
    constructor(props) {
      super(props);
      this.input = React.createRef();
    }

    handleSubmit = (event) => {
      alert('A name was submitted: ' + this.input.current.value);
      event.preventDefault();
    }
    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Name:
            <input type="text" ref={this.input} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

Event Handling

- **Event Handling**: Handle events in React.

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

---

8. Styling in React

Inline Styles

- **Inline Styles**: Apply styles directly to elements.

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

CSS Modules

- **CSS Modules**: Scoped CSS by automatically creating unique class names.

```
import styles from './Button.module.css';

function Button() {
  return <button className={styles.error}>Error Button</button>;
}
```

Styled Components

- **Styled Components**: Use tagged template literals to style components.

```
import styled from 'styled-components';

const Button = styled.button`
  color: palevioletred;
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;
```

CSS-in-JS

- **CSS-in-JS**: Write CSS directly in JavaScript.

```
const styles = {
  container: {
    backgroundColor: 'lightblue',
    padding: '10px',
  },
};

function MyComponent() {
  return <div style={styles.container}>Hello World!</div>;
}
```

---

9. Routing

React Router

- **React Router**: Declarative routing for React.

```
  import { BrowserRouter as Router, Route, Switch } from 'react-router-
dom';

  function App() {
    return (
      <Router>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
        </Switch>
      </Router>
    );
  }
```

Route Configuration

- **Route Configuration**: Define routes in a configuration object.

```
  const routes = [
    { path: '/', component: Home },
    { path: '/about', component: About },
    { path: '/contact', component: Contact },
  ];

  function App() {
    return (
      <Router>
        <Switch>
          {routes.map((route, index) => (
            <Route key={index} path={route.path}
component={route.component} />
          ))}
        </Switch>
      </Router>
    );
  }
```

Navigation

- **Navigation**: Navigate between routes.

```
import { Link } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
```

---

10. Testing

Jest

- **Jest**: JavaScript testing framework.

```
test('adds 1 + 2 to equal 3', () => {
  expect(1 + 2).toBe(3);
});
```

React Testing Library

- **React Testing Library**: Testing utilities for React.

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Unit Testing

- **Unit Testing**: Test individual components.

```
import { render, screen } from '@testing-library/react';
import Button from './Button';

test('renders button with text', () => {
  render(<Button text="Click me" />);
  const buttonElement = screen.getByText(/Click me/i);
  expect(buttonElement).toBeInTheDocument();
});
```

Integration Testing

- **Integration Testing**: Test interactions between components.

```
import { render, screen, fireEvent } from '@testing-library/react';
import App from './App';

test('clicking button changes text', () => {
  render(<App />);
  const buttonElement = screen.getByText(/Click me/i);
  fireEvent.click(buttonElement);
  const changedText = screen.getByText(/Clicked/i);
  expect(changedText).toBeInTheDocument();
});
```

---

11. Deployment

Create React App

- **Create React App**: Set up a modern web app by running one command.

```
npx create-react-app my-app
cd my-app
npm start
```

Environment Variables

- **Environment Variables**: Configure environment-specific settings.

```
REACT_APP_API_URL=https://api.example.com
```

Deployment Strategies

- **Deployment**: Deploy your React app to various platforms.
  - **Netlify**: `npm run build` and deploy the `build` folder.
  - **Vercel**: `vercel` command to deploy.
  - **GitHub Pages**: Use `gh-pages` package to deploy.

---

12. Tips and Tricks

Debugging

- **React Developer Tools**: Browser extension for debugging React applications.

- **console.log**: Use `console.log` for basic debugging.

- **Error Boundaries**: Catch and log errors in components.

Best Practices

- **Component Naming**: Use PascalCase for component names.

- **State Management**: Use state for data that changes over time.

- **Props Validation**: Use `prop-types` for validating props.

- **Code Splitting**: Split code to improve performance.

Common Pitfalls

- **Overusing State**: Avoid unnecessary state updates.

-

By Ahmed Baheeg Khorshid

ver 1.0