React Cheat Sheet

A javascript library for building user interfaces.

DEMO: https://s.codepen.io/ericnakagawa/debug/ALxakj
GITHUB: https://github.com/facebook/react
DOCUMENTATION: https://facebook.github.io/react/docs/
CDN: https://cdnjs.com/libraries/react/

Hello World

```
// Import React and ReactDOM
import React from 'react'
import ReactDOM from 'react-dom'

// Render component into the DOM - only once per app
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

Stateless Components

```
// Stateless React Component
const Headline = () => {
  return <h1>React Cheat Sheet</h1>
}

// Component that receives props
const Greetings = (props) => {
  return <p>You will love it {props.name}.</p>
}

// Component must only return ONE element (eg. DIV)
const Intro = () => {
  return (
    <div>
      <Headline />
      <p>Welcome to the React world!</p>
      <Greetings name="Petr" />
    </div>
  )
}

ReactDOM.render(
  <Intro />,
  document.getElementById('root')
);

// Components and Props API - http://bit.ly/react-props
// CodePen Demo: http://bit.ly/react-simple
```

ES6 Class

```
// use class for local state and lifecycle hooks
class App extends React.Component {

  // Core
  constructor(props) {
    // fires before component is mounted
    super(props); // makes this refer to this component
    this.state = {date: new Date()}; // set state
  }
  render() {
    return (
      <h1>
        It is {this.state.date.toLocaleTimeString()}.
      </h1>
    )
  }

  // Lifecycle Methods
  componentWillMount() {
   // fires immediately before the initial render
  }
  componentDidMount() {
   // fires immediately after the initial render
  }
  componentWillReceiveProps() {
   // fires when component is receiving new props
  }
  shouldComponentUpdate() {
   // fires before rendering with new props or state
  }
  componentWillUpdate() {
   // fires immediately before rendering
   // with new props or state
  }
  componentDidUpdate() {
   // fires immediately after rendering with new P or S
  }
  componentWillUnmount() {
   // fires immediately before component is unmounted
   // from DOM (removed)
  }
}
      // CodePen Demo: http://bit.ly/react-es6-class
```

Conditional Rendering

```
// conditional rendering of elements and CSS class
render() {
  const {isLoggedIn, username} = this.state;
  return (
    <div className={`login ${isLoggedIn ? 'is-in'
: 'is-out'}`}>
      {
        !!isLoggedIn ?
        <p>Logged in as {username}.</p>
        :
        <p>Logged out.</p>
      }
    </div>
  )
}
// CodePen Demo: http://bit.ly/react-if-statements
```

Tools and Resources

```
// Create React App
   http://bit.ly/react-app
// React Dev Tools for Chrome
   http://bit.ly/react-dev-tools
// React Code Snippets for Visual Studio Code
   http://bit.ly/react-es6-snippets-for-vscode
// Babel for Sublime Text 3
   http://bit.ly/babel-sublime
```

React is an open source, front-end, JavaScript library for building user interfaces or UI components like Vue.js, It gives us the ability to create components, layouts etc in our application. In this article we will go through all the fundamentals of Reactjs in this React Cheat Sheet.

Installation

Using react in our application is quite easy as we can add it using CDN or by using the CLI to install it from npm.

To add React using the CDN, add this script tags in your html

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```

Or you can install it using NPM:

```
npm install react react-dom --save
```

Using create-react-app

Create React App is a tool that gives you a massive head start when building **React apps.** It gives us the ability to scaffold a new react project with no configuration. We can install this globally on our local machine by running this command on our terminal:

```
npm install -g create-react-app
```

The `-g` command will install it globally on our local machine.

With this installed, we can now scaffold a react project using this command:

```
create-react-app <name of project>
```

When the setup is completed, we can now move into the project and then download the projects dependencies

```
cd <name of project>
npm install --save
```

After install, to server your application, run `npm start` on your terminal.

React DOM

To setup a simple react DOM, we can import ReactDOM, this is what we will use for rendering.

```
import React from "react";
import ReactDOM from "react-dom";

//define a template
const warning = <h1>Hello,I'm Sunil</h1>;

// ReactDOM.render(root node, mounting point)
ReactDOM.render(warning, document.getElementById("root"));
```

- The `ReactDOM.render()` function takes two arguments, HTML code and an HTML element where the code will be mounted.

Functional Component

This is otherwise known as a stateless component which is just a plain javascript function which takes props as an argument and returns a **react** element:

```
import React from 'react';

const Button = () =>
    <button> Apply</button>

export default Button;
```

Now to use this component, we can do this:

```
import React from 'react';

const Button = ({ onClick, className = 'button', children  }) =>
    <button
        onClick={ onClick }
        className={ className }
        type='button'
    >
        { children }
    </button>

export default Button;
```

Class Component

A Class component acts like a function that receives props, but that function also considers a private internal state as additional input that controls the returned JSX.

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

class MyComponent extends Component {
    render() {
        return (
            <div className="main">
                <h1>Helo Devas</h1>
            </div>
        );
    }
}

export default MyComponent;
```

We can pass in some states:

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

class MyComponent () extends Compnent {
    constructor ( props ) {
    super(props);
    this.state = { message: 'Helo Devas' }
    };
```

```
    render() {
        return (
            <div className="main">
                <h1>{ this.state.message }</h1>
            </div>
        );
    }
}

export default MyComponent;
```

Lifecycle Hooks

React component passes through 3 phases which are Mounting, Updating and Unmounting. When a component is about to mounted, React calls 4 built-in-methods:

- Constructor()
- getDerivedStateFromProps()
- render()
- ComponentDidMount()

Mounting Phases

- `Constructor()`

This method called before anything else in the component, when the component is initiated, and it is the natural place to set up the initial state and other initial values. This method passes a prop as a parameter and always start by calling super(props) before setting any state or anything else.

```
class Footer extends React.Component {
constructor(props) {
    super(props);
    this.state = {name: "Sunil"};
  }
  render() {
    return (
      <h1>My name is {this.state.name}</h1>
    );
  }
}

ReactDOM.render(<Footer />, document.getElementById('root'));
```

- `getDerivedStateFromProps()`

The method gets called before rendering elements in the DOM. It is invoked after a component is instantiated as well as when it receives new props.

```
class Footer extends React.Component {
constructor(props) {
```

```
    super(props);
    this.state = {name: "Sunil"};
  }
static getDerivedStateFromProps(props, state) {
    return {name: props.favcol };
  }

  render() {
    return (
      <h1>My name is {this.state.name}</h1>
    );
  }
}

ReactDOM.render(<Footer />, document.getElementById('root'));
```

- `Render()`

This method outputs the defined HTML into the DOM. This is a required method.

```
class Footer extends React.Component {
  render() {
    return (
      <h1>This template will be rendered using the render function</h1>
    );
  }
}

ReactDOM.render(<Footer />, document.getElementById('root'));
```

- `ComponentDidMount()`

This method gets called immediately after the component is rendered. This is the best place to write statements that requires that the component is already placed in the DOM.

```
class Footer extends React.Component {
constructor(props) {
    super(props);
    this.state = {name: "Sunil"};
  }

  componentDidMount() {
    // Everything here runs after the component has been mounted
  }
  render() {
    return (
      <h1>My name is {this.state.name}</h1>
    );
  }
}

ReactDOM.render(<Footer />, document.getElementById('root'));
```

Updating Phase

The component updates whenever there is a change in the component state or props. Some react built-in method gets called when the component is in this state.

- `getDerivedStateFromProps` :This method gets called immediately a component is updated. This basically does the same thing as the method in the mounting phase.
- `ShouldComponentUpdate` : This method returns a boolean(True or False) which specifies whether React should continue with the rendering or not.

```
shouldComponentUpdate() {
        return true;
    }
```

- `render` :This method gets called when the component is updated. It re-renders the HTML to the DOM with the new values:

```
render() {
        return (
          <h1>This is component is changed</h1>
        );
      }
    }

    ReactDOM.render(<Footer />, document.getElementById('root'));
```

- `getSnapshotBeforeUpdate` : This method gives you the ability to have access to the props and state before the component is updated.

```
getSnapshotBeforeUpdate(prevProps, prevState) {
        // get acces to the prepious state here
      }
```

- `ComponentDidUpdate` : This method gets called after the component has been updated.

```
componentDidUpdate() {
        // do something gere.
       // log the presents state of the component
      }
```

Unmounting Phase

This is a state where react removes a component from the DOM. This phase comes with a `componentWillUnmount` built-in method. The method gets called when the component is about to be removed:

```
componentWillUnmount() {
    alert("Component has been removed");
```

```
  }
```

Props

Props is a concept used in passing data from one component to another. basically it is used for data communication:

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

class App extends Component {
    render() {
        return (
            <div className="app">
                <p>My App {this.props.name}</p>
            </div>
        );
    }
}

//passing the data into the component
class Index extends Component {
    render() {
        return (
            <div className="app">
                <App name="Sunil"/>
            </div>
        );
    }
}
```

```
export default Index;
```

React Map

We can iterate through items using the `map` method. Just like you could use it in Vanilla js, we can have an array of items and then use the map method:

```
let test = [1,2,3,4,5,6];
const numberList = test.map(number=>console.log(number))
```

We can also use it in our react component like this:

```
function App() {
  const people = ['Wisdom', 'Ekpot', 'Sunil','Nirav'];

  return (
    <ul>
      {people.map(person => <Person key={person} name={person} />)}
    </ul>
  );
}
```

Here we are passing it as an array to the component.

Events

Just like any other framework or library, we have the ability to bind event listeners to our template, this events listen to methods defined. In React, we could define a click event like this:

```
function App() {

function logSomething() {
    console.log(`Hello i'm sunil`)
  }

return (
    <div>
      <button onClick={logSomething}>Submit</button>
    </div>
  );
}
```

We can also use the `change` event listeners too on input fields:

```
function App() {

function detectChange() {
    console.log(`Changing`)
```

```
  }

return (
    <div>
      <input type="text" name="myInput" onChange={detectChange} />
    </div>
  );
}
```

State

State is basically storing of data. We can store objects, arrays, strings and then use them in our react components. To use data stored in the state, we can use `this` keyword

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

class App extends Component {
    constructor(props) {
        super(props);
        this.state = {messages: 0};
    }

    render() {
        return (
            <div className="app">
                <p>My messages: {this.state.messages}</p>
            </div>
        );
    }
}
```

```
export default App;
```

React HMR

The hot module reload retains the application state which is lost during a full reload. It saves compilation time as it only updates what was changed and not the entire application:

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import MyComponent from './MyComponent';

ReactDOM.render( <MyComponent />, document.getElementById('root') );

if (module.hot) {
    module.hot.accept();
}
```

React Router

To handling routing in react, we have to install the react-router using NPM:

```
npm i --save react-router-dom
```

To route to a component, we can use the `<Route />` tag which takes the path and the component we routing to as an attribute:

```
import {
  BrowserRouter,
  Route
} from 'react-router-dom'

const Hello = () => <h1>Hello world!</h1>

const App = () => (
  <BrowserRouter>
    <div>
      <Route path="/hello" component={Hello} />
    </div>
  </BrowserRouter>
)
```

React State Hooks

This is basically a state management system. To use this, we have to import `useState` from react. Let's write a simple method which will increment the value of a state when a button is clicked:

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

function Example() {
```

```
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

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

React for Beginners (2021)

React Basics

What is React, really?

- React is officially defined as a "JavaScript library for creating user interfaces," but what does that really mean?
- React is a library, made in JavaScript and which we code in JavaScript, to build great applications that run on the web.

What do I need to know for React?

- In other words, you do need to have a basic understanding of JavaScript to become a solid React programmer.
- The most basic JavaScript concepts you should be familiar with are variables, basic data types, conditionals, array methods, functions, and ES modules.
- How do I learn all of these JavaScript skills? Check out the comprehensive guide to learn all of the JavaScript you need for React.

If React was made in JavaScript, why don't we just use JavaScript?

- While React was written in JavaScript, which was built from the ground up for the express purpose of building web applications and gives us tools to do so.
- JavaScript is a 20+ year old language which was created for adding small bits of behavior to the browser through scripts and was not designed for creating complete applications.
- In other words, while JavaScript was used to create React, they were created for very different purposes.

Can I use JavaScript in React applications?

- Yes! Any valid JavaScript code can be included within your React applications.
- You can use any browser or window API, such as geolocation or the fetch API.

- Also, since React (when it is compiled) runs in the browser, you can perform common JavaScript actions like DOM querying and manipulation.

How to Create React Apps

Three different ways to create a React application

1. Putting React in an HTML file with external scripts
2. Using an in-browser React environment like CodeSandbox
3. Creating a React app on your computer using a tool like Create React App

What is the best way to create a React app?

- Which is the best approach for you? The best way to create your application depends on what you want to do with it.
- If you want to create a complete web application that you want to ultimately push to the web, it is best to create that React application on your computer using a tool like Create React App.
- If you are interested in creating React apps on your computer, check out the complete guide to using Create React App.
- The easiest and most beginner-friendly way to create and build React apps for learning and prototyping is to use a tool like CodeSandbox. You can create a new React app in seconds by going to react.new!

JSX Elements

JSX is a powerful tool for structuring applications

- **JSX** is meant to make create user interfaces with JavaScript applications easier.
- JSX borrows its syntax from the most widely used programming language: HTML
- As a result, JSX is a powerful tool to structure our applications.

- The code example below is the most basic example of a React element which displays the text "Hello World"

```
<div>Hello React!</div>
```

> To be displayed in the browser, React elements need to be rendered (using ReactDOM.render())

How JSX differs from HTML

- We can write valid HTML element in JSX, but what differs slightly is the way some attributes are written.
- Attributes that consist of multiple words are written in the camel-case syntax (i.e. `className`) and have different names than standard HTML (`class`).

```
<div id="header">
  <h1 className="title">Hello React!</h1>
</div>
```

- The reason JSX has this different way of writing attributes is because it is actually made using JavaScript functions (more on this later).

JSX must have a trailing slash if it is made of one tag

- Unlike standard HTML, elements like `input`, `img`, or `br` must close with a trailing forward slash for it to be valid JSX.

```
<input type="email" />// <input type="email"> is a syntax error
```

JSX elements with two tags must have a closing tag

- Elements that should have two tags, such as `div`, `main` or `button`, must have their closing, second tag in JSX, otherwise it will result in a syntax error.

```
<button>Click me</button>// <button> or </button> is a syntax error
```

How JSX elements are styled

- Inline styles are written differently as well as compared to plain HTML.
- Inline styles must not be included as a string, but within an object.
- Once again, the style properties that we use must be written in the camel-case style.

```
<h1 style={{ color: "blue", fontSize: 22, padding: "0.5em 1em" }}>
  Hello React!
</h1>;
```

> Style properties that accept pixel values (like width, height, padding, margin, etc), can use integers instead of strings. For example, fontSize: 22 instead of fontSize: "22px"

JSX can be conditionally displayed

- New React developers may be wondering how it is beneficial that React can use JavaScript code.
- One simple example if that to conditionally hide or display JSX content, we can use any valid JavaScript conditional, like an if statement or switch statement.

```
const isAuthUser = true;

if (isAuthUser) {
  return <div>Hello user!</div>
} else {
  return <button>Login</button>
}
```

- Where are we returning this code? Within a React component, which we will cover in a later section.

JSX cannot be understood by the browser

- As mentioned above, JSX is not HTML, but composed of JavaScript functions.

- In fact, writing `<div>Hello React</div>` in JSX is just a more convenient and understandable way of writing code like the following:

```
React.createElement("div", null, "Hello React!")
```

- Both pieces of code will have the same output of "Hello React".
- To write JSX and have the browser understand this different syntax, we must use a **transpiler** to convert JSX to these function calls.
- The most common transpiler is called **Babel.**

Components

What are React components?

- Instead of just rendering one or another set of JSX elements, we can include them within React **components**.
- Components are created using what looks like a normal JavaScript function, but is different in that it returns JSX elements.

```
function Greeting() {
  return <div>Hello React!</div>;
}
```

Why use React components?

- React components allow us to create more complex logic and structures within our React application than we would with JSX elements alone.
- Think of React components as our custom React elements that have their own functionality.
- As we know, functions allow us to create our own functionality and reuse it where we like across our application.
- Components are reusable wherever we like across our app and as many times as we like.

Components are not normal JavaScript functions

- How would we render or display the returned JSX from the component above?

```
import React from 'react';
import ReactDOM from 'react-dom';

function Greeting() {
  return <div>Hello React!</div>;
}

ReactDOM.render(<Greeting />, document.getElementById("root"));
```

- We use the `React` import to parse the JSX and `ReactDOM` to render our component to a **root element** with the id of "root."

What can components return?

- Components can return valid JSX elements, as well as strings, numbers, booleans, the value `null`, as well as arrays and fragments.
- Why would we want to return `null`? It is common to return `null` if we want a component to display nothing.

```
function Greeting() {
  if (isAuthUser) {
    return "Hello again!";
  } else {
    return null;
  }
}
```

- Another rule is that JSX elements must be wrapped in one parent element. Multiple sibling elements cannot be returned.
- If you need to return multiple elements, but don't need to add another element to the DOM (usually for a conditional), you can use a special React component called a fragment.
- Fragments can be written as `<></>` or when you import React into your file, with `<React.Fragment></React.Fragment>`.

```
function Greeting() {
  const isAuthUser = true;

  if (isAuthUser) {
    return (
      <>
        <h1>Hello again!</h1>
        <button>Logout</button>
      </>
    );
  } else {
    return null;
  }
}
```

> Note that when attempting to return a number of JSX elements that are spread over multiple lines, we can return it all using a set of parentheses () as you see in the example above.

Components can return other components

- The most important thing components can return is other components.
- Below is a basic example of a React application contained with in a component called `App` that returns multiple components:

```
import React from 'react';
import ReactDOM from 'react-dom';

import Layout from './components/Layout';
import Navbar from './components/Navbar';
import Aside from './components/Aside';
import Main from './components/Main';
import Footer from './components/Footer';

function App() {
  return (
    <Layout>
      <Navbar />
      <Main />
      <Aside />
      <Footer />
    </Layout>
  );
}
```

```
ReactDOM.render(<App />, document.getElementById('root'));
```

- What is powerful about this is that we are using the customization of components to describe what they are (i.e. Layout) and their function in our application. This tells us how they should be used just by looking at their name.
- Additionally, we are using the power of JSX to compose these components. In other words, to use the HTML-like syntax of JSX to structure them in an immediately understandable way (i.e. the Navbar is at the top of the app, the Footer at the bottom, etc).

JavaScript can be used in JSX using curly braces

- Just as we can use JavaScript variables within our components, we can use them directly within our JSX as well.
- There are a few core rules to using dynamic values within JSX, however.
- JSX can accept any primitive values (strings, booleans, numbers), but it will not accept plain objects.
- JSX can also include expressions that resolve to these values.
- For example, conditionals can be included within JSX using the ternary operator, since it resolves to a value.

```
function Greeting() {
  const isAuthUser = true;

  return <div>{isAuthUser ? "Hello!" : null}</div>;
}
```

Props

Components can be passed values using props

- Data passed to components in JavaScript are called **props**

- Props look identical to attributes on plain JSX/HTML elements, but you can access their values within the component itself
- Props are available in parameters of the component to which they are passed. Props are always included as properties of an object

```
ReactDOM.render(
  <Greeting username="John!" />,
  document.getElementById("root")
);

function Greeting(props) {
  return <h1>Hello {props.username}</h1>;
}
```

Props cannot be directly changed

- Props must never be directly changed within the child component.
- Another way to say this is that props should never be **mutated**, since props are a plain JavaScript object

```
// We cannot modify the props object:function Header(props) {
  props.username = "Doug";

  return <h1>Hello {props.username}</h1>;
}
```

> Components are consider pure functions. That is, for every input, we should be able to expect the same output. This means we cannot mutate the props object, only read from it.

Special props: the children prop

- The **children** prop is useful if we want to pass elements / components as props to other components
- The children prop is especially useful for when you want the same component (such as a Layout component) to wrap all other components.

```
function Layout(props) {
  return <div className="container">{props.children}</div>;
}

function IndexPage() {
  return (
    <Layout>
      <Header />
      <Hero />
      <Footer />
    </Layout>
  );
}

function AboutPage() {
  return (
    <Layout>
      <About />
      <Footer />
    </Layout>
  );
}
```

- The benefit of this pattern is that all styles applied to the Layout component will be shared with its child components.

Lists and Keys

Iterate over arrays in JSX using map

- How do we displays lists in JSX using array data?
- Use the **.map()** function to convert lists of data (arrays) into lists of elements.

```
const people = ["John", "Bob", "Fred"];
const peopleList = people.map((person) => <p>{person}</p>);
```

.map() can be used for components as well as plain JSX elements.

```
function App() {
  const people = ["John", "Bob", "Fred"];

  return (
    <ul>
```

```
      {people.map((person) => (
        <Person name={person} />
      ))}
    </ul>
  );
}

function Person({ name }) {
// we access the 'name' prop directly using object destructuringreturn <p>This person's na
me is: {name}</p>;
}
```

The importance of keys in lists

- Each React element within a list of elements needs a special **key prop**
- Keys are essential for React to be able to keep track of each element that is being iterated over with the .map() function
- React uses keys to performantly update individual elements when their data changes (instead of re-rendering the entire list)
- Keys need to have unique values to be able to identify each of them according to their key value

```
function App() {
  const people = [
    { id: "Ksy7py", name: "John" },
    { id: "6eAdl9", name: "Bob" },
    { id: "6eAdl9", name: "Fred" },
  ];

  return (
    <ul>
      {people.map((person) => (
        <Person key={person.id} name={person.name} />
      ))}
    </ul>
  );
}
```

State and Managing Data

What is state?

- **State** is a concept that refers to how data in our application changes over time.
- The significance of state in React is that it is a way to talk about our data separately from the user interface (what the user sees).
- We talk about state management, because we need an effective way to keep track of and update data across our components as our user interacts with it.
- To change our application from static HTML elements to a dynamic one that the user can interact with, we need state.

Common examples of using state

- We need to manage state often when our user wants to interact with our application.
- When a user types into a form, we keep track of the form state in that component.
- When we fetch data from an API to display to the user (i.e. posts in a blog), we need to save that data in state.
- When we want to change data that a component is receiving from props, we use state to change it instead of mutating the props object.

Introduction to React hooks with useState

- The way to "create" state is React within a particular component is with the `useState` hook.
- What is a hook? It is very much like a JavaScript function, but can only be used in a React function component at the top of the component.
- We use hooks to "hook into" certain features and useState gives us the ability to create and manage state.
- `useState` is an example of a core React hook that comes directly from the React library: `React.useState`.

```
import React from 'react';

function Greeting() {
```

```
  const state = React.useState("Hello React");

  return <div>{state[0]}</div>// displays "Hello React"}
```

- How does `useState` work? Like a normal function, we can pass it a starting value (i.e. "Hello React").
- What is returned from useState is an array. To get access to the state variable and its value, we can use the first value in that array: `state[0]`.
- There is a way to improve how we write this, however. We can use array destructuring to get direct access to this state variable and call it what we like, i.e. `title`.

```
import React from 'react';

function Greeting() {
  const [title] = React.useState("Hello React");

  return <div>{title}</div>// displays "Hello React"}
```

- What if we want to allow our user to update the greeting they see?
- If we include a form, a user can type in a new value. However, we need a way to update the initial value of our title.

```
import React from "react";

function Greeting() {
  const [title] = React.useState("Hello React");

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" />
    </div>
  );
}
```

- We can do so with the help of the second element in the array that useState returns. It is a setter function, to which we can pass whatever value we want the new state to be.

- In our case, we want to get the value that is typed into the input when a user is in the process of typing. We can get it with the help of React events.

What are events in React?

- Events are ways to get data about a certain action that a user has performed in our app.
- The most common props used to handle events are `onClick` (for click events), `onChange` (when a user types into an input), and `onSubmit` (when a form is submitted.
- Event data is given to us by connecting a function to each of these props listed (there are many more to choose from than these three).
- To get data about the event when our input is changed, we can add `onChange` on input and connect it to a function that will handle the event. This function will be called `handleInputChange`:

```
import React from "react";

function Greeting() {
  const [title] = React.useState("Hello React");

  function handleInputChange(event) {
    console.log("input changed!", event);
  }

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" onChange={handleInputChange} />
    </div>
  );
}
```

> Note that in the code above, a new event will be logged to the browser's console whenever the user types into the input

- Event data is provided to us as an object with many properties which are dependent upon the type of event.

Updating state with useState

- To update state with useState, we can use the second element that useState returns to us in its array.
- This element is a function that will allow us to update the value of the state variable (the first element)
- Whatever we pass to this setter function when we call it will be put in state.

```
import React from "react";

function Greeting() {
  const [title, setTitle] = React.useState("Hello React");

  function handleInputChange(event) {
    setTitle(event.target.value);
  }

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" onChange={handleInputChange} />
    </div>
  );
}
```

- Using the code above, whatever the user types into the input (the text comes from `event.target.value`) will be put in state using `setTitle` and displayed within the `h1` element.
- What is special about state and why it must be managed with a dedicated hook like useState is because a state update (such as when we call `setTitle` causes a re-render.

> A re-render is when a certain component renders or is displayed again based off the new data. If our components weren't re-rendered when data changed, we would never see the app's appearance change at all!