A React cheat sheet.

Tags

Cheat Sheet

JSX

JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code). It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript

```
import logo from "./logo.svg";
import "./App.css";
function App() {
  const element = <h1>Hello, world!</h1>;
  return <div>{element}</div>;
}
export default App;
```

Props

In ReactJS, the props are a type of object where the value of attributes of a tag is stored. The word “props” implies “properties”, and its working functionality is quite similar to HTML attributes. Basically, these props components are read-only components.

```
function Introduction(props) {
  return <h1>Hello, I’m {props.name}!</h1>;
}
function Introduction({ name }) {
  return <h1>Hello, I’m {name}!</h1>;
}
const element = <Introduction name="The Doctor" />;
```

State

What is state? React has another special built-in object called state, which allows components to create and manage their own data. So unlike props, components cannot pass data with state, but they can create and manage it internally.

From Component State to Hooks

From Component State to Hooks

We're going to start with a super simple counter.

Out of the box, it doesn't have a lot going on.

```
class Counter extends Component {
  render() {
    return (
      <main className="Counter">
        <p className="count">0</p>
        <section className="controls">
          <button>Increment</button>
          <button>Decrement</button>
          <button>Reset</button>
        </section>
      </main>
    );
  }
}
```

Let's get it wired up as a fun warmup exercise.

Getting the Basic Component Wired Up

We'll start with a constructor method that sets the component state.

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

We'll use that state in the component.

```
render() {
  const { count } = this.state;

  return (
    <main className="Counter">
      <p className="count">{count}</p>
      <section className="controls">
        <button>Increment</button>
        <button>Decrement</button>
        <button>Reset</button>
      </section>
    </main>
  );
}
```

Alright, now we'll implement the methods to incrementm, decrement, and reset the count.

```
increment() {
  this.setState({ count: this.state.count + 1 });
}

decrement() {
  this.setState({ count: this.state.count - 1 });
}
```

```
reset() {
  this.setState({ count: 0 });
}
```

We'll add those methods to the buttons.

```
<button onClick={this.increment}>Increment</button>
<button onClick={this.decrement}>Decrement</button>
<button onClick={this.reset}>Reset</button>
```

We need to bind those event listeners because everything is terrible.

```
constructor(props) {
  super(props);
  this.state = {
    count: 3,
  };

  this.increment = this.increment.bind(this);
  this.decrement = this.decrement.bind(this);
  this.reset = this.reset.bind(this);
}
```

Component State Pop Quiz

Asyncronous Updates and Queuing

Okay, let's say we refactored `increment()` as follows:

```
increment() {
  this.setState({ count: this.state.count + 1 });
  this.setState({ count: this.state.count + 1 });
  this.setState({ count: this.state.count + 1 });
  console.log(this.state.count);
}
```

Two questions:

1. What will be logged to the console?
2. What will the new value be?

Using a Function as an Argument

`this.setState` also takes a function. This means we could refactor `increment()` as follows.

```
this.setState((state) => {
  return { count: state.count + 1 };
});
```

If we wanted to show off, we could use destructuring to make it evening cleaner.

```
increment() {
  this.setState(({ count }) => {
    return { count: count + 1 };
  });
}
```

There are some potentially cool things we could do here. For example, we could add some logic to our component.

(Live Coding Starts Here)

```
this.setState((state, props) => {
  if (state.count >= 10) return;
  return { count: state.count + 1 };
});
```

Let's stay, we wanted to add in a maximum count as a prop.

```
render(<Counter max={10} />, document.getElementById("roo
t"));
```

```
this.setState((state) => {
  if (state.count >= this.props.max) return;
  return { count: state.count + 1 };
});
```

It turns out that we can actually have a second argument in there as well—the `props`.

```
this.setState((state, props) => {
  if (state.count >= props.max) return;
  return { count: state.count + 1 };
});
```

Oh wait—what if we did this three times?

```
increment() {
  this.setState((state, props) => {
    if (state.count >= props.max) return;
    return { count: state.count + 1 };
  });
  this.setState((state, props) => {
    if (state.count >= props.max) return;
    return { count: state.count + 1 };
  });
  this.setState((state, props) => {
    if (state.count >= props.max) return;
    return { count: state.count + 1 };
  });
}
```

Oh, that's interesting.

The other thing we can do is pull that function out of the component. This makes it *way* easier to unit test.

```
const increment = (state, props) => {
  if (state.count >= props.max) return;
  return { count: state.count + 1 };
};
```

```
increment() {
  this.setState(increment);
}
```

Callbacks

`this.setState` takes a second argument in addition to either the object or function. This function is called after the state change has happened.

Here is the simplest possible implementation.

```
this.setState(increment, () => console.log("Callback"));
```

We can also do something like.

```
this.setState(increment, () => console.log(this.state));
```

Here is a simple thing we might choose to do.

```
this.setState(increment, () => (document.title = `Count: ${th
is.state.count}`));
```

Using LocalStorage in a Side Effect

Let's make a little helper function.

```
const getStateFromLocalStorage = () => {
  const storage = localStorage.getItem("counterState");
  if (storage) return JSON.parse(storage);
  return { count: 0 };
};
```

We can then use a callback to set `localStorage` when the state changes.

```
this.setState(increment, () =>
  localStorage.setItem("counterState", JSON.stringify(this.st
ate))
);
```

Let's pull that out along with increment.

```
const storeStateInLocalStorage = () => {
  localStorage.setItem("counterState", JSON.stringify(this.st
ate));
};
```

```
increment() {
  this.setState(increment, storeStateInLocalStorage);
}
```

It doesn't work. It's a bummer. It would be great if the callback function got a copy of the state, but it doesn't. We could wrap it into a function and then pass the state in.

We could handle this a few ways.

We could use an anonymous function and then pass it in as an argument.

```
const storeStateInLocalStorage = (state) => {
  localStorage.setItem("counterState", JSON.stringify(stat
e));
};
```

```
increment() {
  this.setState(increment, () => storeStateInLocalStorage(thi
s.state));
}
```

Alternatively, if we're willing to give up on arrow functions, we can use `bind`.

```
function storeStateInLocalStorage() {
  localStorage.setItem("counterState", JSON.stringify(this.st
ate));
}
```

```
increment() {
  this.setState(increment, storeStateInLocalStorage.bind(thi
s));
}
```

Lastly, we can just put it onto the class component itself.

```
storeStateInLocalStorage() {
  localStorage.setItem('counterState', JSON.stringify(this.st
ate));
}

increment() {
  this.setState(increment, this.storeStateInLocalStorage);
}
```

This is probably your best bet.

Refactoring to Hooks

Hooks are a new pattern that allow us to write a lot less code. Get ready to delete some code.

Let's start by deleting everything but the render method.

```
const Counter = ({ max }) => {
  const [count, setCount] = React.useState(0);

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  const reset = () => setCount(0);

  return (
    <main className="Counter">
      <p className="count">{count}</p>
      <section className="controls">
        <button onClick={increment}>Increment</button>
        <button onClick={decrement}>Decrement</button>
        <button onClick={reset}>Reset</button>
      </section>
    </main>
  );
};
```

So, what *don't* we have to do here?

- We don't have to `bind` anything.
- We dont need a reference to this.
- We don't need a `constructor` at all.

Running Some of Our Previous Experiments

What if we tripled up again?

```
const increment = () => {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);
};
```

Okay, so that makes sense.

It also turns out that `useState` setters can take functions too.

```
const increment = () => {
  setCount((c) => c + 1);
};
```

Unlike using values, using functions also works the same way as it does with `this.setState`.

```
const increment = () => {
  setCount((c) => c + 1);
  setCount((c) => c + 1);
  setCount((c) => c + 1);
};
```

There is an important difference. You get *only* the state in this case. There is no second argument with the props. That said, we have them in scope.

They also do *not* support callback functions like `this.setState`. Later on, we'll use `useEffect` to trigger side effects based on state changes.

Earlier with `this.setState`, we ended up returning `undefined` if our count had hit the max. What if we did something similar here?

```
setCount((c) => {
  if (c >= max) return;
  return c + 1;
});
```

Oh—it explodes after ten. This is core to the difference between how `useState` and `this.setState` works.

With `this.setState`, we're giving the component that object of values that it needs to update. With `useState`, we've got a dedicated function to change a particular piece of state.

How can we fix this?

```
setCount((c) => {
  if (c >= max) return c;
  return c + 1;
});
```

A Brief Introduction to useEffect

We're going to go a bit deeper into `useEffect`, but let's do the high level now.

Use effect allows us to implement some kind of side effect in our component outside of the changes to state and props triggering a new render.

This is useful for a ton of reasons:

- Storing stuff in `localStorage`.
- Making AJAX requests.

Implementing `localStorage`

Let's get the basic set up in place here.

Here is a reminder of that function of getting from `localStorage`.

```
const getStateFromLocalStorage = () => {
  const storage = localStorage.getItem("counterState");
  if (storage) return JSON.parse(storage);
  return { count: 0 };
};
```

We'll read the count property from `localStorage`.

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

Now, we'll register a side effect.

```
React.useEffect(() => {
  localStorage.setItem("counterState", JSON.stringify({ count
}));
}, [count]);
```

Events

Event handlers determine what action is to be taken whenever an event is fired. This could be a button click or a change in a text input.

Essentially, event handlers are what make it possible for users to interact with your React app. Handling events with React elements is similar to handling events on DOM elements, with a few minor exceptions.

```
function ActionLink() {
  const handleClick = (e) => {
    e.preventDefault();
    console.log("The link was clicked.");
  };

  return <button onClick={handleClick}>Click me</button>;
}
```

What are synthetic events in React?

React implements a synthetic events system that brings consistency and high performance to React apps and interfaces. It achieves consistency by normalizing events so that they have the same properties across different browsers and platforms.

A synthetic event is a cross-browser wrapper around the browser’s native event. It has the same interface as the browser’s native event, including `stopPropagation()`

and `preventDefault()`, except the events work identically across all browsers.

It achieves high performance by automatically using event delegation. In actuality, React doesn't attach event handlers to the nodes themselves. Instead, a single event listener is attached to the root of the document. When an event is fired, React maps it to the appropriate component element.

Call an inline function in an onClick event handler

Inline functions allow you to write code for event handling directly in JSX. See the example below:

```
import React from "react";

const App = () => {
  return <button onClick={() => alert("Hello!")}>Say Hello</b
utton>;
};

export default App;
```

This is commonly used to avoid the extra function declaration outside the JSX, although it can be less readable and harder to maintain if the content of the inline function is too much.

Update the state inside an onClick event handler

Let's say your React application requires you to update the local state in an onClick event handler. Here's how to do that:

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

const App = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</
button>
      <button onClick={() => setCount(count - 1)}>Decrement</
button>
    </div>
  );
};

export default App;
```

In the example above, the value of `useState` is modified by the `Increment` and `Decrement` buttons, which have the `setCount`, an updater function inside the `onClick` event handler.

Call multiple functions in an onClick event handler

The onClick event handler also allows you to call multiple functions.

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

const App = () => {
  const [count, setCount] = useState(0);
  const sayHello = () => {
    alert("Hello!");
  };

  return (
    <div>
      <p>{count}</p>
      <button
        onClick={() => {
          sayHello();
          setCount(count + 1);
        }}
      >
        Say Hello and Increment
      </button>
    </div>
  );
};

export default App;
```

Custom components and events in React

When it comes to events in React, only DOM elements are allowed to have event handlers. Take the example of a component called CustomButton with an onClick event. This button wouldn't respond to clicks because of the reason above.

```
import React from "react";

const CustomButton = ({ onPress }) => {
  return (
    <button type="button" onClick={onPress}>
      Click on me
    </button>
  );
};

const App = () => {
  const handleEvent = () => {
    alert("I was clicked");
  };
  return <CustomButton onPress={handleEvent} />;
};
export default App;
```

Life Cycle

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

constructor() static getDerivedStateFromProps() render() componentDidMount()

Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

static getDerivedStateFromProps() shouldComponentUpdate() render() getSnapshotBeforeUpdate() componentDidUpdate()

Unmounting

This method is called when a component is being removed from the DOM:

componentWillUnmount()

Hooks;

useState

useEffect

This means useEffect runs on every render and thus the event handlers will unnecessarily get detached and reattached on each render.

```
window.React.useEffect(() => {
  console.log(`adding listener ${count}`);
  window.addEventListener("click", listener);
});
```

component will unmount

his is the optional cleanup mechanism for effects. Every effect may return a function that cleans up after it. This lets us keep the logic for adding and removing subscriptions close to each other. They’re part of the same effect! the side-effect runs after every rendering.

}

```
window.React.useEffect(() => {
  return () => {
    console.log(`removing listener ${count}`);
    window.removeEventListener("click", listener);
  };
});
```

```
import { useEffect } from "react";
function RepeatMessage({ message }) {
  useEffect(() => {
    const id = setInterval(() => {
      console.log(message);
    }, 2000);
    return () => {
      clearInterval(id);
    };
  }, [message]);
  return <div>I'm logging to console "{message}"</div>;
}
```

useEffect componentDidMount

the side-effect runs once after the initial rendering.

```
import { useEffect } from "react";
function Greet() {
  let name = "Hassan Habib Tahir";
  const message = `Hello, ${name}!`; // Calculates output
  useEffect(() => {
    // Good!
    document.title = `Greetings to ${name}`; // Side-effect!
  }, []);
  return <div>{message}</div>; // Calculates output
}
```

callback is the function containing the side-effect logic. callback is executed right after changes were being pushed to DOM. dependencies is an optional array of dependencies. useEffect() executes callback only if the dependencies have changed between renderings.

```
useEffect(callback[, dependencies]);
```

Component did update

```
import { useEffect } from "react";
function MyComponent({ prop }) {
  const [state, setState] = useState();
  useEffect(() => {
    // Side-effect uses `prop` and `state`
  }, [prop, state]);
  return <div>....</div>;
}
```

useMemo

useMemo is a React Hook that lets you cache the result of a calculation between re-renders. The basic purpose of the useMemo hook is related to the fact that we try to avoid the unnecessary re-rendering of components and props in our program. **Example 1** Let's make a simple application to demonstrate the use of the useMemo hook.

The program below has the following components:

Increment button: starts from 0 and increases the count by 1.

Even num button: starts from 2 and returns even numbers going forward.

An **evenNumDoouble()** function that returns the twice value of the even number.

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

function Counter() {
  const [count, setCount] = useState(0);
  const [evenNum, setEvenNum] = useState(2);

  function evenNumDouble() {
    console.log("double");
    return evenNum * 2;
  }

  return (
    <div>
      <h2>Counter: {count}</h2>
      <h2>Even Numbers: {evenNum}</h2>
      <h2>even Number Double Value: {evenNumDouble()}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</
button>
      <button onClick={() => setEvenNum(evenNum + 2)}>Even Nu
mbers</button>
    </div>
  );
}

export default Counter;
```

Explanation When we click the button Even Numbers, the evenNumDouble() function is called because the state of evenNum is changed.

Clicking the Increment button also renders the evenNumDouble() function, although the count state does not change.

This means that every time the evenNumDouble() function is rendered unnecessarily (on the page), the code becomes less efficient. We will fix this with the useMemo hook below.

```
import React, { useState, useMemo } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  const [evenNum, setEvenNum] = useState(2);
  const memoHook = useMemo(
    function evenNumDouble() {
      console.log("double");
      return evenNum * 2;
    },
    [evenNum]
  );

  return (
    <div>
      <h2>Counter: {count}</h2>
      <h2>Even Numbers: {evenNum}</h2>
      <h2>even Number Double Value: {memoHook}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</
button>
      <button onClick={() => setEvenNum(evenNum + 2)}>Even Nu
mbers</button>
    </div>
  );
}

export default Counter;
```

Explanation In the code above, we set the output of the evenNumDouble() function into a constant memoHook.

This filters the function through the useMemo hook to only check if the specified variable (evenNum in this case) has been changed; only then will this function be rendered.

```
import React, { memo, useState } from "react";
import { useContext } from "react";
import { GrudgeContext } from "./GrudgeContext";
const NewGrudge = memo(({ onSubmit }) => {
  const { addGrudge } = React.useContext(GrudgeContext);
  const [person, setPerson] = useState("");
  const [reason, setReason] = useState("");

  const handleChange = (event) => {
    event.preventDefault();
    addGrudge({ person, reason });
  };

  return (
    <form className="NewGrudge" onSubmit={handleChange}>
      <input
        className="NewGrudge-input"
        placeholder="Person"
        type="text"
        value={person}
        onChange={(event) => setPerson(event.target.value)}
      />
      <input
        className="NewGrudge-input"
        placeholder="Reason"
        type="text"
        value={reason}
        onChange={(event) => setReason(event.target.value)}
      />
      <input className="NewGrudge-submit button" type="submi
t" />
    </form>
  );
});

export default NewGrudge;
```

useCallback

useCallback is a React Hook that lets you cache a function definition between re-renders.

```
const [grudges, dispatch] = useReducer(reducer, initialStat
e);
const addGrudge = useCallback(
  ({ person, reason }) => {
    dispatch({
      type: GRUDGES_ADD,
      payload: {
        person,
        reason,
      },
    });
  },
  [dispatch]
);
```

useRef;

useRef is a React Hook that lets you reference a value that’s not needed for rendering. and create a mutable variable which will not re-render the components

useRef(initialValue) is a built-in React hook that accepts one argument as the initial value and returns a reference (aka ref). A reference is an object having a special property current.

```
import { useRef } from "react";
function MyComponent() {
  const reference = useRef(initialValue);
  const someHandler = () => {
    // Access reference value:
    const value = reference.current;
    // Update reference value:
    reference.current = newValue;
  };
}
```

Updating a reference doesn't trigger a component re-rendering.

```
import { useRef } from "react";
function LogButtonClicks() {
  const countRef = useRef(0);

  const handle = () => {
    countRef.current++;
    console.log(`Clicked ${countRef.current} times`);
  };
  console.log("I rendered!");
  return <button onClick={handle}>Click me</button>;
}
```

Accessing DOM elements

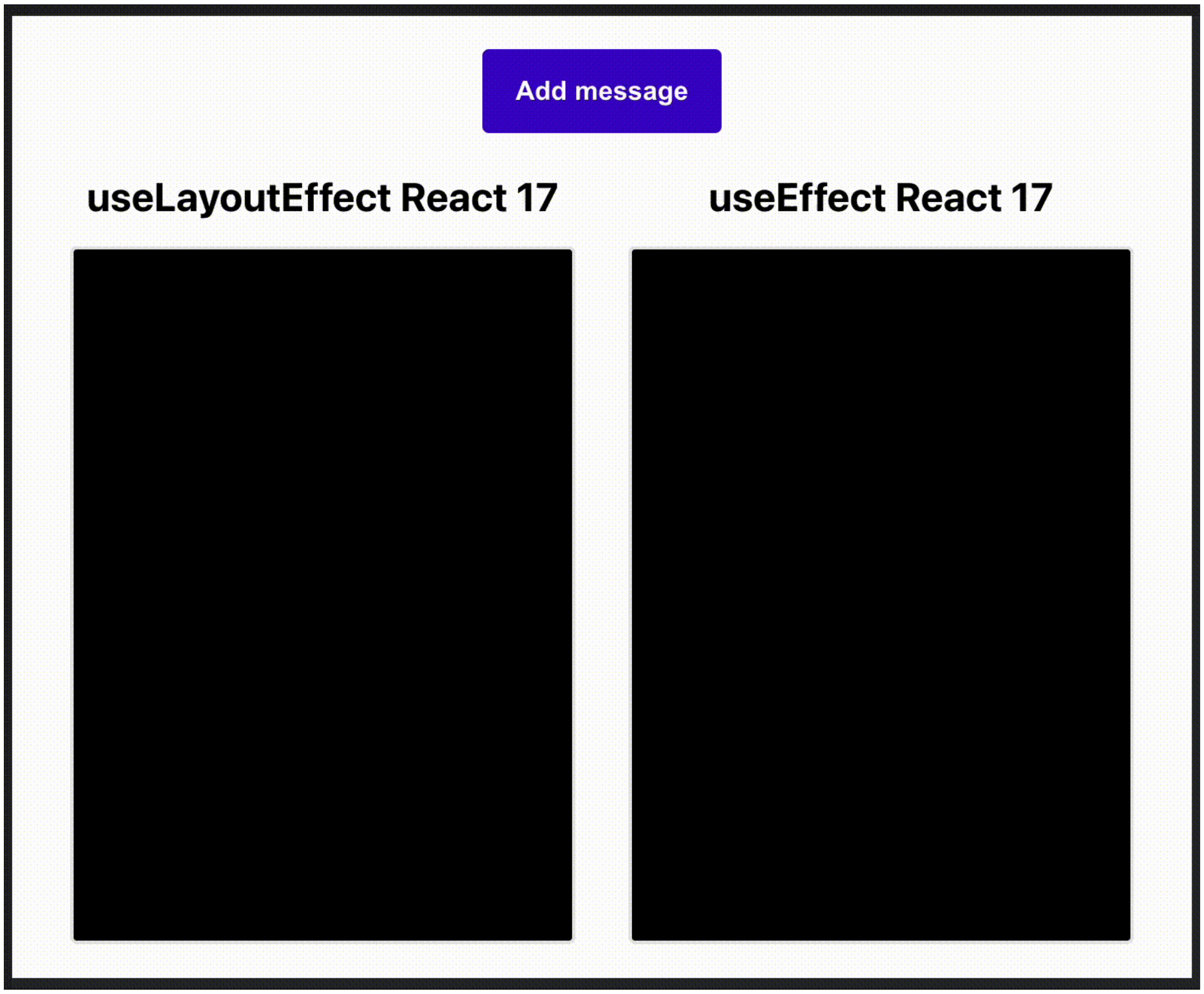
```
import { useRef, useEffect } from "react";
function AccessingElement() {
  const elementRef = useRef();
  useEffect(() => {
    const divElement = elementRef.current;
    console.log(divElement); // logs <div>I'm an element</div
>
  }, []);
  return <div ref={elementRef}>I'm an element</div>;
}
```

useLayoutEffect

useEffect hook serves asynchronously, whereas the useLayoutEffect hook works synchronously;

This runs synchronously immediately after React has performed all DOM mutations. This can be useful if you need to make DOM measurements (like getting the scroll position or other styles for an element) and then make DOM mutations or trigger a synchronous re-render by updating state.

As far as scheduling, this works the same way as componentDidMount and componentDidUpdate. Your code runs immediately after the DOM has been updated, but before the browser has had a chance to "paint" those changes (the user doesn't actually see the updates until after the browser has repainted).



```
import React, { useLayoutEffect } from "react";
const APP = (props) => {
  useLayoutEffect(() => {
    //Do something and either return undefined or a cleanup f
unction
    return () => {
      //Do some cleanup here
    };
  }, [dependencies]);
};
```

Style Your React App

Inline Styles

Inline styles are the most direct away to style any React application.

```
export default function App() {
  return (
    <section
      style={{
        fontFamily: "-apple-system",
        fontSize: "1rem",
        fontWeight: 1.5,
        lineHeight: 1.5,
        color: "#292b2c",
        backgroundColor: "#fff",
        padding: "0 2em",
      }}
    >
      <div
        style={{
          textAlign: "center",
          maxWidth: "950px",
          margin: "0 auto",
          border: "1px solid #e6e6e6",
          padding: "40px 25px",
          marginTop: "50px",
        }}
      >
        <img
          src="https://randomuser.me/api/portraits/women/48.j
pg"
```

```
          alt="Tammy Stevens"
          style={{
            margin: "-90px auto 30px",
            width: "100px",
            borderRadius: "50%",
            objectFit: "cover",
            marginBottom: "0",
          }}
        />
        <div>
          <p
            style={{
              lineHeight: 1.5,
              fontWeight: 300,
              marginBottom: "25px",
              fontSize: "1.375rem",
            }}
          >
            This is one of the best developer blogs on the pl
anet! I read it
            daily to improve my skills.
          </p>
        </div>
        <p
          style={{
            marginBottom: "0",
            fontWeight: 600,
            fontSize: "1rem",
          }}
        >
          Tammy Stevens
          <span style={{ fontWeight: 400 }}> · Front End Deve
loper</span>
        </p>
      </div>
    </section>
```

```
  );
}
```

Plain CSS

Instead of using inline styles, it's common to import a CSS stylesheet to style a component's elements. Writing CSS in a stylesheet is probably the most common and basic approach to styling a React application, but it shouldn't be dismissed so easily.

```
/* src/styles.css */

body {
  font-family: -apple-system, system-ui, BlinkMacSystemFont,
"Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;
  margin: 0;
  font-size: 1rem;
  font-weight: 1.5;
  line-height: 1.5;
  color: #292b2c;
  background-color: #fff;
}
.testimonial {
  margin: 0 auto;
  padding: 0 2em;
}
.testimonial-wrapper {
  text-align: center;
  max-width: 950px;
  margin: 0 auto;
  border: 1px solid #e6e6e6;
  padding: 40px 25px;
  margin-top: 50px;
}
.testimonial-quote p {
  line-height: 1.5;
  font-weight: 300;
  margin-bottom: 25px;
  font-size: 1.375rem;
}
.testimonial-avatar {
  margin: -90px auto 30px;
  width: 100px;
  border-radius: 50%;
  object-fit: cover;
  margin-bottom: 0;
}
.testimonial-name {
  margin-bottom: 0;
  font-weight: 600;
  font-size: 1rem;
}
.testimonial-name span {
  font-weight: 400;
}
```

SASS / SCSS

What is SASS? SASS is an acronym that stands for: Syntactically Awesome Style Sheets. SASS gives us some powerful tools, many of which don't exist in normal CSS stylesheets. It includes features like variables, extending styles, and nesting.

```
/* styles.scss */
npm install node-sass
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }
  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

CSS Modules

CSS modules are another slight alternative to something like CSS or SASS. What is great about CSS modules is that they can be used with either normal CSS or SASS.

```
/* src/styles.module.css */

body {
  font-family: -apple-system, system-ui, BlinkMacSystemFont,
"Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;
  margin: 0;
  font-size: 1rem;
  font-weight: 1.5;
  line-height: 1.5;
  color: #292b2c;
  background-color: #fff;
}

/* styles skipped */

.testimonial-name span {
  font-weight: 400;
}
```

```
import styles from "./styles.module.css";
```

```
export default function App() {
  return (
    <section className={styles.testimonial}>
      <div className={styles["testimonial-wrapper"]}>
        <img
          src="https://randomuser.me/api/portraits/women/48.j
pg"
          alt="Tammy Stevens"
          className={styles["testimonial-avatar"]}
        />
        <div>
          <p className={styles["testimonial-quote"]}>
            This is one of the best developer blogs on the pl
anet! I read it
            daily to improve my skills.
          </p>
        </div>
        <p className={styles["testimonial-name"]}>
          Tammy Stevens
          <span> · Front End Developer</span>
        </p>
      </div>
    </section>
  );
}
```

CSS-in-JS

Similar to how React allowed us to write HTML as JavaScript with JSX, CSS-in-JS has done something similar with CSS.

```
import styled from "styled-components";

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

  &:hover {
    opacity: 0.9;
  }
`;

export default function App() {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );
}
```

Using a Reducer

We could try to get clever here with `useCallback` and `React.memo`, but since we're always replacing the array of grudges, this is never really going to work out.

What if we took a different approach to managing state?

Let's make a new file called `reducer.js`.

```
const reducer = (state = [], action) => {
  return state;
};
```

And then we swap out that `useState` with a `useReducer`.

```
const [grudges, dispatch] = useReducer(reducer, initialStat
e);
```

We're going to create an action type and an action creator.

```
const GRUDGE_ADD = "GRUDGE_ADD";
const GRUDGE_FORGIVE = "GRUDGE_FORGIVE";
```

```
const addGrudge = ({ person, reason }) => {
  dispatch({
    type: GRUDGE_ADD,
    payload: {
      person,
      reason,
    },
  });
};
```

We'll add it to the reducer.

```
const reducer = (state = [], action) => {
  if (action.type === GRUDGE_ADD) {
    return [
      {
        id: id(),
        ...action.payload,
      },
      ...state,
    ];
  }
  return state;
};
```

Forgiveness

Let's make an action creator

```
const forgiveGrudge = (id) => {
  dispatch({
    type: GRUDGE_FORGIVE,
    payload: {
      id,
    },
  });
};
```

We'll also update the reducer here.

```
if (action.type === GRUDGE_FORGIVE) {
  return state.map((grudge) => {
    if (grudge.id === action.payload.id) {
      return { ...grudge, forgiven: !grudge.forgiven };
    }
    return grudge;
  });
}
```

We'll thread through `forgiveGrudge` as `onForgive`.

```
<button onClick={() => onForgive(grudge.id)}>Forgive</button>
```

The Context API

What is Context API?

The React Context API is a way for a React app to effectively produce global variables that can be passed around. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. The above example wasn't *too* bad. But, you can see how it might get a bit out of hand as our application grows.

What if two very different distant cousin components needed the same data?

Modern builds of React allow you to use something called the Context API to make this better. It's basically a way for very different pieces of your application to communicate with each other.

We're going to rip a lot out of `Application.js` and move it to a new file called `GrudgeContext.js` and it's going to look something like this.

```
import React, { useReducer, createContext, useCallback } from
"react";
import initialState from "./initialState";
import id from "uuid/v4";

export const GrudgeContext = createContext();

const GRUDGE_ADD = "GRUDGE_ADD";
const GRUDGE_FORGIVE = "GRUDGE_FORGIVE";

const reducer = (state = [], action) => {
  if (action.type === GRUDGE_ADD) {
    return [
      {
        id: id(),
        ...action.payload,
      },
      ...state,
    ];
  }

  if (action.type === GRUDGE_FORGIVE) {
    return state.map((grudge) => {
      if (grudge.id === action.payload.id) {
        return { ...grudge, forgiven: !grudge.forgiven };
      }
      return grudge;
    });
  }

  return state;
};
```

```
export const GrudgeProvider = ({ children }) => {
  const [grudges, dispatch] = useReducer(reducer, initialStat
e);

  const addGrudge = useCallback(
    ({ person, reason }) => {
      dispatch({
        type: GRUDGE_ADD,
        payload: {
          person,
          reason,
        },
      });
    },
    [dispatch]
  );

  const toggleForgiveness = useCallback(
    (id) => {
      dispatch({
        type: GRUDGE_FORGIVE,
        payload: {
          id,
        },
      });
    },
    [dispatch]
  );

  return (
    <GrudgeContext.Provider value={{ grudges, addGrudge, togg
leForgiveness }}>
      {children}
    </GrudgeContext.Provider>
  );
};
```

Now, `Application.js` looks a lot more slimmed down.

```
import React from "react";

import Grudges from "./Grudges";
import NewGrudge from "./NewGrudge";

const Application = () => {
  return (
    <div className="Application">
      <NewGrudge />
      <Grudges />
    </div>
  );
};

export default Application;
```

Wrapping the Application in Your New Provider

```
ReactDOM.render(
  <GrudgeProvider>
    <Application />
  </GrudgeProvider>,
  rootElement
);
```

That works and it's cool, but it's still missing the point.

Hooking Up the Context API

So, we don't need that pass through on `Grudges` anymore. Let's rip that out completely.

```
import React from "react";
import Grudge from "./Grudge";

const Grudges = ({ grudges = [] }) => {
  return (
    <section className="Grudges">
      <h2>Grudges ({grudges.length})</h2>
      {grudges.map((grudge) => (
        <Grudge key={grudge.id} grudge={grudge} />
      ))}
    </section>
  );
};

export default Grudges;
```

But, we will need to tell it about the grudges so that it can iterate through them.

```
import React from "react";
import Grudge from "./Grudge";
import { GrudgeContext } from "./GrudgeContext";

const Grudges = () => {
  const { grudges } = React.useContext(GrudgeContext);

  return (
    <section className="Grudges">
      <h2>Grudges ({grudges.length})</h2>
      {grudges.map((grudge) => (
        <Grudge key={grudge.id} grudge={grudge} />
      ))}
    </section>
  );
};
```

```
export default Grudges;
```

Individual Grudges

```
import React from "react";
import { GrudgeContext } from "./GrudgeContext";

const Grudge = ({ grudge }) => {
  const { toggleForgiveness } = React.useContext(GrudgeContex
t);

  return (
    <article className="Grudge">
      <h3>{grudge.person}</h3>
      <p>{grudge.reason}</p>
      <div className="Grudge-controls">
        <label className="Grudge-forgiven">
          <input
            type="checkbox"
            checked={grudge.forgiven}
            onChange={() => toggleForgiveness(grudge.id)}
          />{" "}
          Forgiven
        </label>
      </div>
    </article>
  );
};

export default Grudge;
```

Adding a New Grudge with the Context API

In this case, we *just* need the ability to add a grudge.

```
const NewGrudge = () => {
  const [person, setPerson] = React.useState('');
  const [reason, setReason] = React.useState('');
  const { addGrudge } = React.useContext(GrudgeContext);

  const handleSubmit = event => {
    event.preventDefault();

    addGrudge({
      person,
      reason
    });
  };

  return (
    // …
  );
};

export default NewGrudge;

const defaultGrudges = {
  1: {
    id: 1,
    person: name.first(),
    reason: 'Parked too close to me in the parking lot',
    forgiven: false
  },
  2: {
    id: 2,
    person: name.first(),
    reason: 'Did not brew another pot of coffee after drinkin
g the last cup',
    forgiven: false
  }
};
```

```
export const GrudgeProvider = ({ children }) => {
  const [grudges, setGrudges] = useState({});

  const addGrudge = (grudge) => {
    grudge.id = id();
    setGrudges({
      [grudge.id]: grudge,
      ...grudges,
    });
  };

  const toggleForgiveness = (id) => {
    const newGrudges = { ...grudges };
    const target = grudges[id];
    target.forgiven = !target.forgiven;
    setGrudges(newGrudges);
  };

  return (
    <GrudgeContext.Provider
      value={{ grudges: Object.values(grudges), addGrudge, to
ggleForgiveness }}
    >
      {children}
    </GrudgeContext.Provider>
  );
};
```

```
const defaultState = {
  past: [],
  present: [],
  future: [],
};
```

We've broken almost everything. So, let's make this a bit better.

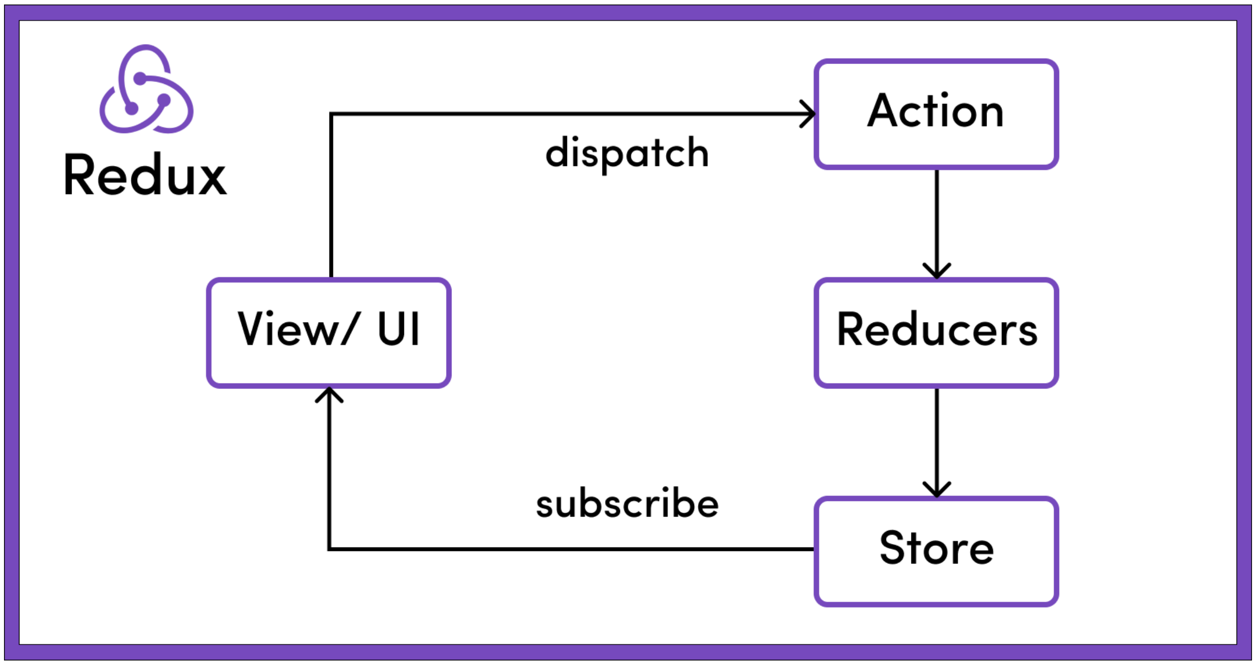
```
const reducer = (state, action) => {
  if (action.type === ADD_GRUDGE) {
    return {
      past: [],
      present: [
        {
          id: uniqueId(),
          ...action.payload,
        },
        ...state.present,
      ],
      future: [],
    };
  }

  if (action.type === FORGIVE_GRUDGE) {
    return {
      past: [],
      present: state.present.filter(
        (grudge) => grudge.id !== action.payload.id
      ),
      future: [],
    };
  }

  return state;
};
```

Why Redux?

State transfer between components is pretty messy in React since it is hard to keep track of which component the data is coming from. It becomes really complicated if users are working with a large number of states within an application.



Redux solves the state transfer problem by storing all of the states in a single place called a store. So, managing and transferring states becomes easier as all the states are stored in the same convenient store. Every component in the application can then directly access the required state from that store.

What is Redux?

Redux is a predictable state container for JavaScript applications. It helps you write apps that behave consistently, run in different environments (client, server, and native), and are easy to test. Redux manages an application's state with a single global object called Store.

Pillars of Redux

These are Redux's main pillars:

Store:

A store is an object that holds the application's state tree. There should only be a single store in a Redux app, as the composition happens at the reducer level. getState() returns the current state of the store.

dispatch()

dispatches an action. It is the only way to update the application state.

```
() => dispatch({ msg: "ADD_SOMETHING" });
```

subscribe() subscribes a change listener to the state. unsubscribe() is useful when you no longer want to call your listener method when the state changes.

Actions:

An action is a plain object that represents an intention to change the state. They must have a property to indicate the type of action to be carried out.

. Actions are payloads of information that send data from your application to your store. . Any data, whether from UI events or network callbacks, needs to eventually be dispatched as actions. . Actions must have a type field, indicating the type of action being performed.

Reducers:

Reducers are pure functions that specify how the application's state changes in response to actions sent to the store. . Actions only describe what happened, not how the application's state changes. . A reducer is a function that accepts the current state and action, and returns a new state with the action performed. . combineReducers() utility can be used to combine all the reducers in the app into a single index reducer which makes maintainability much easier

Web Application with Redux

Create Store

1. Create a store in the index.js file

```
//import redux from redux;
// create Store;
```

```
import { createStore } from "redux";
import { composeWithDevTools } from "redux-devtools-extensio
n";
import rootReducer from "./rootReducer";
// create Store
// plance function
// all functions are defined in the store;
const store = createStore(rootReducer, composeWithDevTools
());

export default store;
```

Main

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import * as serviceWorker from "./serviceWorker";
import { Provider } from "react-redux";
import store from "./redux/store";
ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Reducer

```
import { combineReducers } from "redux";
import {
  GET_ALL_PRODUCT,
  GET_NUMBER_CART,
  ADD_CART,
  DECREASE_QUANTITY,
  INCREASE_QUANTITY,
  DELETE_CART,
} from "./types";

const initProduct = {
  numberCart: 0,
  Carts: [],
  _products: [],
};

function todoProduct(state = initProduct, action) {
  switch (action.type) {
    case GET_ALL_PRODUCT:
      return {
        ...state,
        _products: action.payload,
      };
    case GET_NUMBER_CART:
      return {
        ...state,
      };
    case ADD_CART:
      if (state.numberCart == 0) {
        let cart = {
          id: action.payload.id,
          quantity: 1,
          name: action.payload.name,
          image: action.payload.image,
          price: action.payload.price,
        };
        state.Carts.push(cart);
      } else {
        let check = false;
        state.Carts.map((item, key) => {
          if (item.id == action.payload.id) {
            state.Carts[key].quantity++;
            check = true;
          }
        });
        if (!check) {
          let _cart = {
            id: action.payload.id,
            quantity: 1,
            name: action.payload.name,
            image: action.payload.image,
            price: action.payload.price,
          };
          state.Carts.push(_cart);
        }
      }
      return {
        ...state,
        numberCart: state.numberCart + 1,
      };
    case INCREASE_QUANTITY:
      state.numberCart++;
      state.Carts[action.payload].quantity++;

      return {
        ...state,
      };
    case DECREASE_QUANTITY:
      let quantity = state.Carts[action.payload].quantity;
      if (quantity > 1) {
        state.numberCart--;
        state.Carts[action.payload].quantity--;
      }

      return {
        ...state,
      };
    case DELETE_CART:
      let quantity_ = state.Carts[action.payload].quantity;
      return {
        ...state,
        numberCart: state.numberCart - quantity_,
        Carts: state.Carts.filter((item) => {
          return item.id != state.Carts[action.payload].id;
        }),
      };
    default:
      return state;
  }
}
const ShopApp = combineReducers({
  _todoProduct: todoProduct,
});
export default ShopApp;
```

Actions

```
export const INCREASE_QUANTITY = "INCREASE_QUANTITY";
export const DECREASE_QUANTITY = "DECREASE_QUANTITY";
export const GET_ALL_PRODUCT = "GET_ALL_PRODUCT";
export const GET_NUMBER_CART = "GET_NUMBER_CART";
export const ADD_CART = "ADD_CART";
export const UPDATE_CART = "UPDATE_CART";
export const DELETE_CART = "DELETE_CART";

export const actFetchProductsRequest = () => {
  return (dispatch) => {
    return callApi("products", "GET", null).then((res) => {
      dispatch(GetAllProduct(res.data));
    });
  };
};

/*GET_ALL_PRODUCT*/
export function GetAllProduct(payload) {
  return {
    type: "GET_ALL_PRODUCT",
    payload,
  };
}

/*GET NUMBER CART*/
export function GetNumberCart() {
  return {
    type: "GET_NUMBER_CART",
  };
}

export function AddCart(payload) {
  return {
    type: "ADD_CART",
    payload,
  };
}
export function UpdateCart(payload) {
  return {
    type: "UPDATE_CART",
    payload,
  };
}
export function DeleteCart(payload) {
  return {
    type: "DELETE_CART",
    payload,
  };
}
```

```
export function IncreaseQuantity(payload) {
  return {
    type: "INCREASE_QUANTITY",
    payload,
  };
}
export function DecreaseQuantity(payload) {
  return {
    type: "DECREASE_QUANTITY",
    payload,
  };
}
```

components

```
import React from "react";
import "./navbar.scss";
import logo from "../../assets/logo.png";
import { Link } from "react-router-dom";
import { useSelector } from "react-redux";
const Navbar = () => {
  const getReducer = useSelector((state) => state._todoProduc
t);
  const { numberCart } = getReducer;
  console.log(getReducer, "getReducer Navbar ---->");
  return (
    <div className="parent_nav">
      <div>
        <Link to="/">
          {" "}
          <div>
            <img
              src={logo}
              alt=""
              style={{
                width: "30%",
                height: "30%",
              }}
            />
          </div>
        </Link>
      </div>
      <Link to="/cart">
        <div className="cart_container">
          <div>
            <svg
              xmlns="http://www.w3.org/2000/svg"
              style={{
                fontWeight: "bolder",
              }}
              width="30"
              height="30"
              fill="#F2246B"
              className="bi bi-minecart"
              viewBox="0 0 16 16"
            >
              <path d="M4 15a1 1 0 1 1 0-2 1 1 0 0 1 0 2zm0 1
a2 2 0 1 0 0-4 2 2 0 0 0 0 4zm8-1a1 1 0 1 1 0-2 1 1 0 0 1 0 2
zm0 1a2 2 0 1 0 0-4 2 2 0 0 0 0 4zM.115 3.18A.5.5 0 0 1 .5 3h
15a.5.5 0 0 1 .491.592l-1.5 8A.5.5 0 0 1 14 12H2a.5.5 0 0 1-.
491-.408l-1.5-8a.5.5 0 0 1 .106-.411zm.987.82 1.313 7h11.17l
1.313-7H1.102z" />
            </svg>
          </div>
          <div
            style={{
              color: "#F2246B",
            }}
          >
            {numberCart}
          </div>
        </div>
      </Link>
    </div>
  );
};

export default Navbar;
```

```
import Axios from "axios";
import React from "react";
import { useDispatch, useSelector } from "react-redux";
import { actFetchProductsRequest, AddCart } from "../../stor
e/action";
export const Products = () => {
  const dispatch = useDispatch();
  const getReducer = useSelector((state) => state._todoProduc
t);
  const { _products } = getReducer;
  React.useEffect(() => {
    (async () => {
      dispatch(actFetchProductsRequest());
    })();
  }, []);
  const CartHandler = (product) => {
    dispatch(AddCart(product));
  };
  return (
    <div className="mt-3">
      <div class="container">
        <div class="row">
          {_products &&
            _products.map((item, key) => (
              <div class="col col-lg-3" key={key}>
                <div className="pb-1">
```

```
                  <div className="product-item bg-light mb-
4">
                    <div className="product-img position-rela
tive overflow-hidden">
                      <img
                        className="img-fluid w-100 "
                        style={{
                          height: "250px",
                        }}
                        alt=""
                        src={item.image}
                        alt="website template image"
                      />
                    </div>
                    <div className="text-center py-4">
                      <a
                        className="h6 text-decoration-none te
xt-truncate"
                        href="https://www.free-css.com/free-c
ss-templates"
                      >
                        {item.name}
                      </a>
                      <div className="d-flex align-items-cent
er justify-content-center mt-2">
                        <h5>${item.price}</h5>
                        <h6 className="text-muted ml-2">
                          <del>$123.00</del>
                        </h6>
                      </div>
                      <div>
                        <button
                          type="button"
                          className="btn btn-outline-dark"
                          onClick={() => CartHandler(item)}
                        >
```

```
                            Add to cart
                          </button>
                        </div>
                      </div>
                    </div>
                  </div>
                </div>
              ))}
          </div>
        </div>
      </div>
    );
  };
```

Difference Between Context API and Redux

In other way in redux you can set the data in one components in other components have right to acces in data. ..// On the other hand, Context deals with them as they happen on the component level that's main you should to share data in components like props.

Difference betwen context api and props

props work with one direction main data share in one component,context api you can share data with 3 or 4 components