React.JS Documentation

Getting Started with React:

What is React? React apps are made out of *components.* A component is a piece of the UI (user interface) that has its own logic and appearance and it can be as small as a button, or as large as an entire page.

Installation of React:

To start working with React, the user must install a react app and it can be done be typing in the terminal the following

`npx create-react-app`

Also react must be imported at top of each file page by writing:

```
import React from 'react':
```

Component:

Components are one of the core concepts of React. They are the foundation upon which you build user interfaces (UI), which makes them the perfect place for React. We have 2 types of components, functional component that are defined as JavaScript and they take properties (props) as their input that is responsible to print it on the screen, and class component that extends from (React. Components).

Defining a Component:

```
function Button () {
 return <button>Click</button>
};
function Profile () {
 return (
<Button/>
 ); };
Export default Profile;
```

Where Profile is the name of the component (must be always Uppercase).
Components can be nested into each other as in the above , the Button function was nested into the Profile function.
After each component, it must be exported so it can be used in another file by writing the following:

```
export default "Component Name";
```

Or it can be typed another way:

```
export default function "Component Name"( ) { };
```

In order to use is in another file, it must be imported at the beginning of the file page:

```
import "Component Name" from './file name';
```

From the above example, if Button was in another file called button.js,
the code will be the following:

```
import React from 'react';
import Button from './button.js'
function Profile () {
 return (
<Button/>
 ); };
Export default Profile;
```

Writing Markup with JSX:

JSX is a syntax extension for JavaScript that let us write HTML-like markup inside a JavaScript file. Although there are other ways to write components.
In JSX the syntax is written as:

```
const panda = ‘red’;
```

Where panda is the name and red is the value.

JSX elements can be nested inside each other's:

```
const example = (<a href = “….”
                   <h1>...</h1>>
                   </a>);
```

h1 was nested inside of the a.

Writing JSX must be wrapped always in one outcome (parent div):

```
function example () {
  return(
    <div>
      <p>...</p>
    </div>
    )
}
```

This code will work with no problems, while writing without div or <> won't work.

Regarding the self-closing tags, we must add backslash in their tags like
 and <img/>

Inside JSX expressions, we can access variables:

```
function Avatar () {
 const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
 const description = 'Gregorio Y. Zara';
return (

<img
 className="avatar"
 src={avatar}
 alt={description}/> ); };
export default Avatar;
```

Avatar is a style in CSS
const description is a String
src = {avatar} is the content of src in Avatar
alt = {description} is the content of description in Avatar

Passing Properties (props) to Components:

Props are the information that is passed to a JSX tag like className, src, alt, width, height.

```
function Avatar () {
 return (
 <img
 className="avatar"
 src=https://i.imgur.com/1bX5QH6.jpg"
 alt="Lin Lanying"
 width= {100}
 height= {100} />);}

function Profile ({person, size}) {
 return (
 <Avatar
 person= {{name: 'Lin Lanying', imgId: '1'}}
 size = {100} />);};
```

Conditional Rendering:

We can write conditions inside components but must be before the return statement:

```
function example () {
let number = 10;
if (number <= 5) {
 return <p>The number is {number}</p>;
}
else {
return <p>Number still wrong</p>;
}
```

The condition is a typical if-else statement, but we can use another condition called Ternary Operator that works same as if-else statement.

```
{number <= 5 ? "The number is {number}" : "Number is still
wrong"}
```

number <= 5 is the condition

1st string execute if true
2nd string execute if false

Rendering Lists:

We will often want to display multiple similar components from a collection of data. We can use the JavaScript array methods to manipulate an array of data

Rendering Data from arrays:

Rendering components from an array using JavaScript's `map():`

We have an array with the following data:

```
const people = [{
 id: 0,
 name: 'Creola Katherine Johnson',
 profession: 'mathematician',
}, {
 id: 1,
 name: 'Mario José Molina-Pasquel Henríquez',
 profession: 'chemist',
}, {
 id: 2,
 name: 'Mohammad Abdus Salam',
 profession: 'physicist',
}, {
 id: 3,
 name: 'Percy Lavon Julian',
 profession: 'chemist',
},];
```

We can use .map() in order in make a new array called listItems with only names in it:

```
const listItems = people.map(person => <li>{person}</li>);
```

And then we need to render it in order to appear on the screen:

```
return <ul>{listItems}</ul>;
```

Rendering components from an array using JavaScript's `filter():`

Same as the example before but without .map(), we can use the .filter() and insert them in a new array called chemists :

```
const chemists = people.filter (person => person.profession === 'chemist');
```

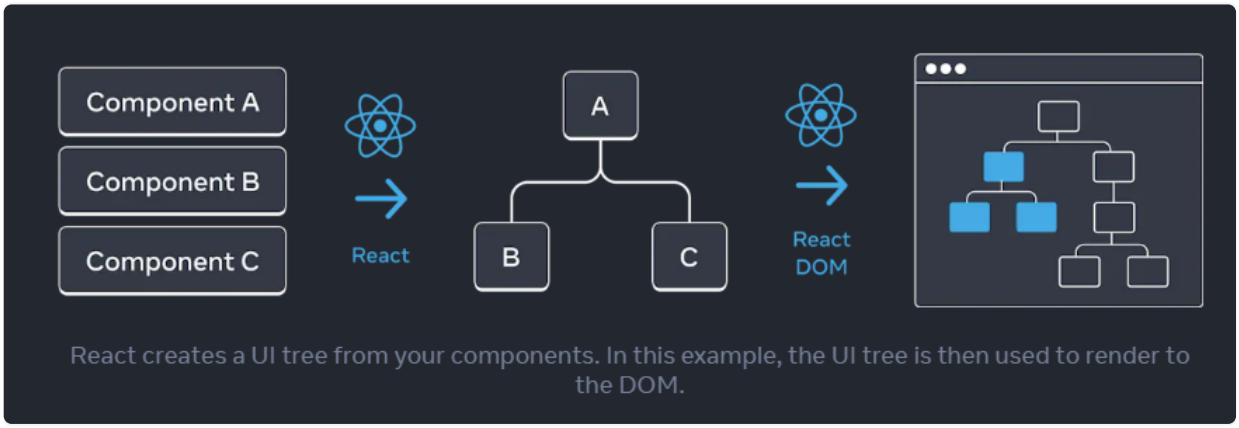
The array chemists will contain from the people array only names that have the profession is equal to chemist

Understanding the UI as a Tree:

React, and many other UI libraries, model UI as a tree. Thinking of the app as a tree is useful for understanding the relationship between components. This understanding will help the user to debug future concepts like performance and state management.

The UI as a Tree:

Trees are a relationship model between items and UI is often represented using tree structures. For example, browsers use tree structures to model HTML (DOM) and CSS (CSSOM). Mobile platforms also use trees to represent their view hierarchy.



React creates a UI tree from your components. In this example, the UI tree is then used to render to the DOM.

Adding Interactivity:

In React, data that changes over time is called state. We can add state to any component, and update it as needed, most of the time somethings on the screen update in response to user input it can be clicking or anything the user might change on screen.

Responding to events

React lets you add *event handlers* to your JSX. Event handlers are your own functions that will be triggered in response to user interactions like clicking, hovering, focusing on form inputs, and so on. Built-in components like `<button>` only support built-in browser events like `onClick`. However, we can also create your own components, and give their event handler props any application-specific names that you like.
Example:

```
export default function App() {
  return (
    <Toolbar
      onPlayMovie={() => alert('Playing!')}
      onUploadImage={() => alert('Uploading!')}
    />
  );
}
function Toolbar({ onPlayMovie, onUploadImage }) {
  return (
    <div>
      <Button onClick={onPlayMovie}>
        Play Movie
      </Button>
      <Button onClick={onUploadImage}>
        Upload Image
      </Button>
    </div>
  );
}
function Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );
}
```

A function Button was declared that have onClick and childern as props for it.
This is the JSX code that defines the structure of the `Button` component. It returns a `<button>` element with an `onClick` event handler set to the `onClick` prop passed to the

component. The content of the button is represented by the `{children}` expression, which renders any child elements or text content passed to the `Button` component.

State: a component's memory:

Components need to "remember" things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called *state.* You can add state to a component with a `useState` Hook. *Hooks* are special functions that let your components use React features (state is one of those features). The `useState` Hook lets you declare a state variable. It takes the initial state and returns a pair of values: the current state, and a state setter function that lets you update it.

```
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
  const [index, setIndex] = useState(0);
  const [showMore, setShowMore] = useState(false);
  const hasNext = index < sculptureList.length - 1;

  function handleNextClick() {
    if (hasNext) {
      setIndex(index + 1);
    } else {
      setIndex(0);
    }
  }
  function handleMoreClick() {
    setShowMore(!showMore);
  }
  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNextClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
```

```
        </h2>
        <h3>
          ({index + 1} of {sculptureList.length})
        </h3>
        <button onClick={handleMoreClick}>
          {showMore ? 'Hide' : 'Show'} details
        </button>
        {showMore && <p>{sculpture.description}</p>}
        <img
          src={sculpture.url}
          alt={sculpture.alt}
        />
      </>
    );
  }
```

Render and commit:

1. **Triggering** a render (delivering the diner's order to the kitchen)
2. **Rendering** the component (preparing the order in the kitchen)
3. **Committing** to the DOM (placing the order on the table)

State as a snapshot :

React state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render.

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

export default function Form() {
  const [to, setTo] = useState('Alice');
  const [message, setMessage] = useState('Hello');
  function handleSubmit(e) {
    e.preventDefault();
    setTimeout(() => {
      alert(`You said ${message} to ${to}`);
    }, 5000);
  }
  return (
```

```
      <form onSubmit={handleSubmit}>
        <label>
          To:{' '}
          <select
            value={to}
            onChange={e => setTo(e.target.value)}>
            <option value="Alice">Alice</option>
            <option value="Bob">Bob</option>
          </select>
        </label>
        <textarea
          placeholder="Message"
          value={message}
          onChange={e => setMessage(e.target.value)}
        />
        <button type="submit">Send</button>
      </form>
    );
  }
```

Updating objects in state:

We need to create a new one (or make a copy of an existing one), and then update the state to use that copy.

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

export default function Form() {
  const [person, setPerson] = useState({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });
  function handleNameChange(e) {
    setPerson({
      ...person,
      name: e.target.value
```

```
    });
  }
  function handleTitleChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        title: e.target.value
      }
    });
  }
  function handleCityChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        city: e.target.value
      }
    });
  }
  function handleImageChange(e) {
    setPerson({
      ...person,
      artwork: {
        ...person.artwork,
        image: e.target.value
      }
    });
  }
  return (
    <>
      <label>
        Name:
        <input
          value={person.name}
          onChange={handleNameChange}
        />
      </label>
      <label>
        Title:
```

```
          <input
            value={person.artwork.title}
            onChange={handleTitleChange}
          />
        </label>
        <label>
          City:
          <input
            value={person.artwork.city}
            onChange={handleCityChange}
          />
        </label>
        <label>
          Image:
          <input
            value={person.artwork.image}
            onChange={handleImageChange}
          />
        </label>
        <p>
          <i>{person.artwork.title}</i>
          {' by '}
          {person.name}
          <br />
          (located in {person.artwork.city})
        </p>
        <img
          src={person.artwork.image}
          alt={person.artwork.title}
        />
      </>
    );
  }
```

Updating arrays in state:

Just like objects, we can update arrays in states.

	avoid (mutates the array)	prefer (returns a new array)
adding	`push`, `unshift`	`concat`, `[...arr]` spread syntax (example)
removing	`pop`, `shift`, `splice`	`filter`, `slice` (example)
replacing	`splice`, `arr[i] = ...` assignment	`map` (example)
sorting	`reverse`, `sort`	copy the array first (example)

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

const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];
export default function BucketList() {
  const [list, setList] = useState(
    initialList
  );
  function handleToggle(artworkId, nextSeen) {
    setList(list.map(artwork => {
      if (artwork.id === artworkId) {
        return { ...artwork, seen: nextSeen };
      } else {
        return artwork;
      }
    }));
  }
  return (
    <>
      <h1>Art Bucket List</h1>
      <h2>My list of art to see:</h2>
      <ItemList
        artworks={list}
        onToggle={handleToggle} />
    </>
  );
}
```

```
function ItemList({ artworks, onToggle }) {
  return (
    <ul>
      {artworks.map(artwork => (
        <li key={artwork.id}>
          <label>
            <input
              type="checkbox"
              checked={artwork.seen}
              onChange={e => {
                onToggle(
                  artwork.id,
                  e.target.checked
                );
              }}
            />
            {artwork.title}
          </label>
        </li>
      ))}
    </ul>
  );
}
```

Inserting into an array

Sometimes, you may want to insert an item at a particular position that's neither at the beginning nor at the end. To do this, you can use the `...` array spread syntax together with the `slice()` method. The `slice()` method lets you cut a "slice" of the array. To insert an item, you will create an array that spreads the slice *before* the insertion point, then the new item, and then the rest of the original array.

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

let nextId = 3;
const initialArtists = [
  { id: 0, name: 'Marta Colvin Andrade' },
  { id: 1, name: 'Lamidi Olonade Fakeye'},
```

```
    { id: 2, name: 'Louise Nevelson'},
];
export default function List() {
  const [name, setName] = useState('');
  const [artists, setArtists] = useState(
    initialArtists
  );
  function handleClick() {
    const insertAt = 1; // Could be any index
    const nextArtists = [
      // Items before the insertion point:
      ...artists.slice(0, insertAt),
      // New item:
      { id: nextId++, name: name },
      // Items after the insertion point:
      ...artists.slice(insertAt)
    ];
    setArtists(nextArtists);
    setName('');
  }
  return (
    <>
      <h1>Inspiring sculptors:</h1>
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
      <button onClick={handleClick}>
        Insert
      </button>
      <ul>
        {artists.map(artist => (
          <li key={artist.id}>{artist.name}</li>
        ))}
      </ul>
    </>
  );
}
```

Updating objects inside arrays:

Objects are not *really* located “inside” arrays. They might appear to be “inside” in code, but each object in an array is a separate value, to which the array “points”. This is why you need to be careful when changing nested fields like `list[0]`. Another person’s artwork list may point to the same element of the array.

When updating nested state, you need to create copies from the point where you want to update, and all the way up to the top level.

Example:

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

let nextId = 3;
const initialList = [
  { id: 0, title: 'Big Bellies', seen: false },
  { id: 1, title: 'Lunar Landscape', seen: false },
  { id: 2, title: 'Terracotta Army', seen: true },
];
export default function BucketList() {
  const [myList, setMyList] = useState(initialList);
  const [yourList, setYourList] = useState(
    initialList
  );
  function handleToggleMyList(artworkId, nextSeen) {
    const myNextList = [...myList];
    const artwork = myNextList.find(
      a => a.id === artworkId
    );
    artwork.seen = nextSeen;
    setMyList(myNextList);
  }
  function handleToggleYourList(artworkId, nextSeen) {
    const yourNextList = [...yourList];
    const artwork = yourNextList.find(
      a => a.id === artworkId
    );
    artwork.seen = nextSeen;
    setYourList(yourNextList);
  }
  return (
```

```
      <>
        <h1>Art Bucket List</h1>
        <h2>My list of art to see:</h2>
        <ItemList
          artworks={myList}
          onToggle={handleToggleMyList} />
        <h2>Your list of art to see:</h2>
        <ItemList
          artworks={yourList}
          onToggle={handleToggleYourList} />
      </>
    );
  }
  function ItemList({ artworks, onToggle }) {
    return (
      <ul>
        {artworks.map(artwork => (
          <li key={artwork.id}>
            <label>
              <input
                type="checkbox"
                checked={artwork.seen}
                onChange={e => {
                  onToggle(
                    artwork.id,
                    e.target.checked
                  );
                }}
              />
              {artwork.title}
            </label>
          </li>
        ))}
      </ul>
    );
  }
```

Write concise update logic with Immer:

Immer is a JavaScript library that simplifies the process of working with immutable state by allowing you to write code that looks like mutable code, while ensuring that immutability is maintained behind the scenes. While Immer itself is not specific to React, it is commonly used in React applications, especially in combination with state management libraries like Redux or when working with complex state structures.

Immer provides a convenient way to update immutable state by using a concept called "drafts." With Immer, you can work with a mutable draft of your state, making changes to it as if it were mutable, and then Immer automatically produces an immutable copy of the updated state. This makes it easier to write and reason about state updates, especially in scenarios where the state is deeply nested or has complex structures.

From the above example, we can use:

```
updatePerson (draft => { draft.artwork.city = ‘Lagos’;});
```

To try Immer, we run in the terminal `npm install use-immer` to add it as a dependency then we import it by `import {useImmer} from ‘use-immer`’

```
import { useImmer } from 'use-immer';

export default function Form() {
 const [person, updatePerson] = useImmer({
 name: 'Niki de Saint Phalle',
 artwork: {
 title: 'Blue Nana',
 city: 'Hamburg',
 image: 'https://i.imgur.com/Sd1AgUOm.jpg',
 }
 });
 function handleNameChange(e) {
 updatePerson(draft => {
 draft.name = e.target.value;
 });
 }
 function handleTitleChange(e) {
 updatePerson(draft => {
```

```
  draft.artwork.title = e.target.value;
  });
  }
  function handleCityChange(e) {
  updatePerson(draft => {
  draft.artwork.city = e.target.value;
  });
  }
  function handleImageChange(e) {
  updatePerson(draft => {
  draft.artwork.image = e.target.value;
  });
  }
```

Preventing Default Behavior:

```
function Signup () {
 return (
 <form onSubmit= {e => {
 e.preventDefault ();
 alert('Submitting!');
 }}>
 <input />
 <button>Send</button>
 </form>
 );};
export default Signup;
```

The `e.preventDefault` statement prevents the default browser behavior for the few events that have it.

State: A Component's Memory:

The state in React refers to an object that represents the internal state of a component. It allows components to keep track of information that can change over time, such as user input, UI state, or data fetched from an API. State is a fundamental concept in React because it enables components to be dynamic and interactive.

The useState hook provides 2 things:

. A **state variable** to retain the data between renders.

. A **state setter function** to update the variable & trigger React to render the component again.

To add a **state variable,** we import the useState at the top of the file:

```
import React, {useState} from 'react';
function Counter () {
 // Declare a state variable named "count" and a function to update it, "setCount"
 const [count, setCount] = useState (0);
 return (
 <div>
 <p>You clicked {count} times</p>
 {/* When the button is clicked, update the "count" state */}
 <button onClick= {() => setCount (count + 1)}>
 Click me
 </button></div>);}
export default Counter;
```

Rendering and Commit:

Before the components are displayed on the screen, they must be rendered by React.

Triggering a render:

In order to trigger a render, there are 2 reasons for it: it's the component's **initial render** and the component's **state has been updated**.

Initial Render:

When the app starts, the initial render must be triggered by 2 ways, either when creating a react app it is applied by default to the file App.js and if they weren't available we can write the following:

```
import {createRoot} from 'react-dom/client';
const root = createRoot (document.getElementById ('…');
root.render ();
```

```
import Image from './Image.js';
import {createRoot} from 'react-dom/client';
const root = createRoot (document.getElementById ('root'))
root.render (<Image />);
```

We imported Image from file Image.js and imported the createRoot function from the react-dom/client.

Declared a variable called root with createRoot function is called with an argument document.getElementById('root'), which presumably represents a DOM element with the id 'root'. This function returns a Root object.

The .render method of the Root object is then called with the <Image /> JSX element as an argument. This renders the Image component into the specified root element and printing the image on the screen.

States as a Snapshot:

State behaves like a snapshot. Setting it won't change the state of the variable that is already had, instead it will trigger a re-render.

Setting state triggers renders:

In this example when pressing the 'send' button the setIsSent will become true and a message will appear on the screen.

```

import { useState } from 'react';
export default function Form() {
 const [isSent, setIsSent] = useState(false);
 const [message, setMessage] = useState('Hi!');
 if (isSent) {
 return <h1>Your message is on its way! </h1>
 }
 return (
 <form onSubmit={(e) => {
 e.preventDefault ();
 setIsSent (true);
```

```
    sendMessage (message);
    }}>
    <textarea
    placeholder="Message"
    value={message}
    onChange={e => setMessage(e.target.value)}
    />
    <button type="submit">Send</button>
    </form>
    );
  }
  function sendMessage(message) {
    // ...
  }
```

Queueing a Series of State Updates:

Setting a state variable will queue another render. But sometimes you might want to perform multiple operations on the value before queueing the next render.

React batches state updates:

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

export default function Counter() {
  const [number, setNumber] = useState(0);
  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
        setNumber(number + 1);
      }}>+3</button>
    </>
  )
}
```

This example will only increment one time and not 3 times. In order to make in increment 3 times the following code must be written:

```jsx
export default function Counter() {
  const [number, setNumber] = useState(0);
  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(n => n + 1);
        setNumber(n => n + 1);
        setNumber(n => n + 1);
      }}>+3</button>
    </>
  )
}
```

Here, n => n + 1 is called an updater function. When you pass it to a state setter:

. React queues this function to be processed after all the other code in the event handler has run.

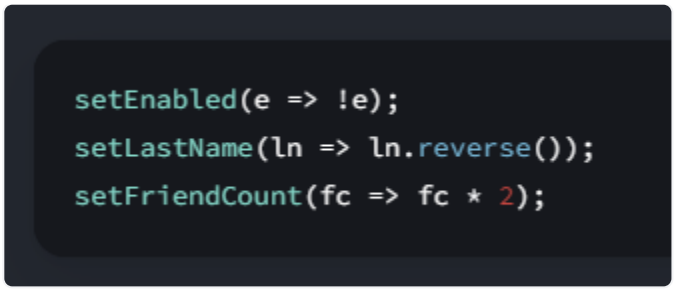
. During the next render, React goes through the queue and gives you the final updated state.

queued update	n	returns
n => n + 1	0	0 + 1 = 1
n => n + 1	1	1 + 1 = 2
n => n + 1	2	2 + 1 = 3

React stores 3 as the final result and returns it from the

Naming Conventions:

It's common to name the updater function argument by the first letters of the corresponding state variable:



Updating Objects in State:

State can hold any kind of JavaScript value, including objects, but we shouldn't change objects that we hold in the React state directly. Instead, when we want to update an object, we need to create a new one (or make a copy of an existing one), and then set the state to use that copy.

Mutation in React refers to the direct modification of state or props within a component.
const [x , setX] = useState (0);

We can store now in the setX any value we want:
setX(2);

The x state changed from 0 to 2, but the number 0 itself did not change. It's not possible to make any changes to the built-in primitive values like numbers, strings, and Booleans in JavaScript.

```
const [position, setPosition] = useState ({x: 0, y: 0});
```

We can change the contents of the object itself and it is called **Mutation**.

```
position.x = 0;
```

Copying objects with the spread syntax:

Consider the following:

```
setPerson ({
 firstName: e.target.value,
 lastName: person.lastName,
 email: person.email });
```

In the above example, we can use the object spread (...) syntax so that no need to copy every property separately.

```
setPerson({
 ...person, // Copy the old fields
 firstName: e.target.value // But override this one
});
```

Updating Nested objects:

```
const [person, setPerson] = useState({
 name: 'Niki de Saint Phalle',
 artwork: {
 title: 'Blue Nana',
 city: 'Hamburg',
 image: 'https://i.imgur.com/Sd1AgUOm.jpg',
 }
});

```

If we want to update `person.artwork.city` we can use mutation and it becomes like the following:

```
person.artwork.city = ‘New Delhi’;
```

But in React, we treat the state as immutable! In order to change city, we need to produce the new artwork object (pre-populated with data from the previous one), and then produce the new person object which points at the new artwork and it becomes like the following:

```
const nextArtwork = { ...person.artwork, city: 'New Delhi' };
const nextPerson = { ...person, artwork: nextArtwork };
setPerson(nextPerson);
```

Or written as a single function:

```
setPerson({
 ...person, // Copy other fields
```

```
  artwork: { // but replace the artwork
  ...person.artwork, // with the same one
  city: 'New Delhi' // but in New Delhi!
  }
});
```

Reacting to Input with State:

You describe the different states that your components can be in and switch them in response to the user input.

How declarative UI compares to imperative for example:

- When you type something into the form, the “Submit” button **becomes enabled.**
- When you press “Submit”, both the form and the button **become disabled,** and a spinner **appears.**

Think about UI declaratively:

1. **Identify** your component’s different visual states
2. **Determine** what triggers those state changes
3. **Represent** the state in memory using useState
4. **Remove** any non-essential state variables
5. **Connect** the event handlers to set the state

Identify your component’s different visual states

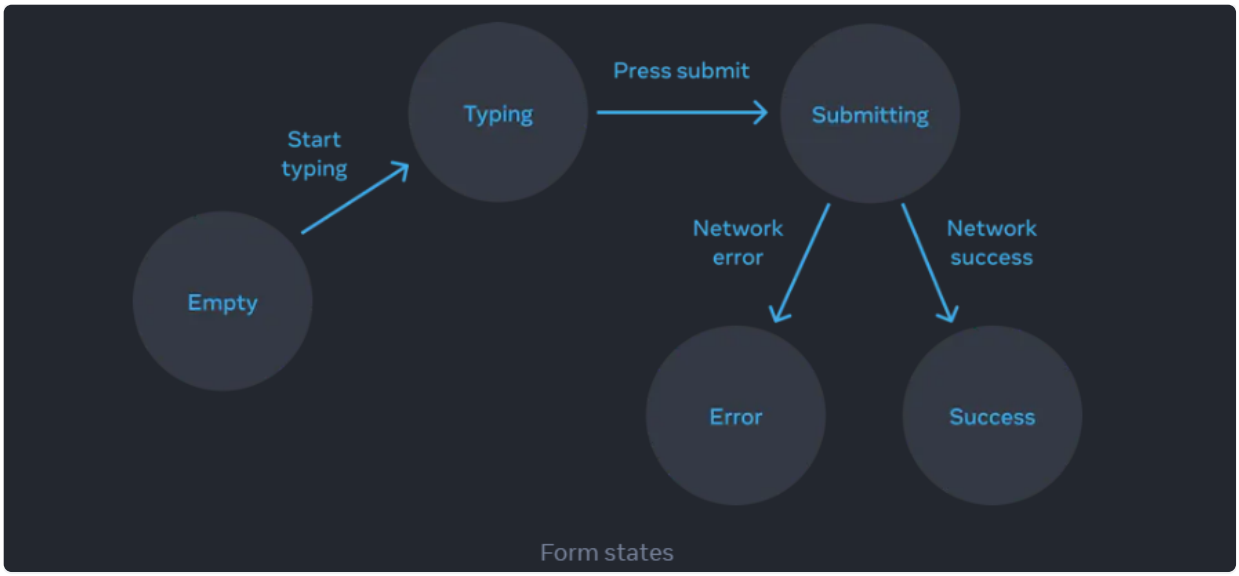
The states that designers might see:

- Empty (**disabled submit button**)
- Typing (**enabled submit button**)
- Submitting (**completely disabled**)
- Success (**thank you message is shown**)
- Error (**same as typing state, but with extra error message**)

Determine what triggers those state changes

Human input such as **clicking button, typing in a field.**

Computer inputs such as **network response arriving , a timeout completing, an image loading.**



Form states

Represent the state in memory with useState

The representation of the visual states in memory is done with useState

```
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
Start by adding enough state to be sure that all the possible visual states are covered
const [isEmpty, setIsEmpty] = useState(true);
const [isTyping, setIsTyping] = useState(false);
const [isSubmitting, setIsSubmitting] = useState(false);
const [isSuccess, setIsSuccess] = useState(false);
const [isError, setIsError] = useState(false);
```

Remove any non-essential state variables

To avoid duplication in the state, you should clean up.

```
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
const [status, setStatus] = useState('typing'); // 'typing', 'submitting', or 'success'
```

Connect the event handlers to set state

Create event handlers that update the state.

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

export default function Form() {
 const [answer, setAnswer] = useState('');
 const [error, setError] = useState(null);
 const [status, setStatus] = useState('typing');

 if (status === 'success') {
 return <h1>That's right!</h1>
 }
 async function handleSubmit(e) {
 e.preventDefault();
 setStatus('submitting');
 try {
 await submitForm(answer);
 setStatus('success');
 } catch (err) {
 setStatus('typing');
 setError(err);
 }
 }
 function handleTextareaChange(e) {
 setAnswer(e.target.value);
 }
 return (
 <>
 <h2>City quiz</h2>
 <p>
 In which city is there a billboard that turns air into
drinkable water?
 </p>
 <form onSubmit={handleSubmit}>
 <textarea
 value={answer}
 onChange={handleTextareaChange}
 disabled={status === 'submitting'}
 />
 <br />
 <button disabled={
```

```
39    answer.length === 0 ||
40    status === 'submitting'
41    }>
42    Submit
43    </button>
44    {error !== null &&
45    <p className="Error">
46    {error.message}
47    </p>
48    }
49    </form>
50    </>
51    );
52  }
53  function submitForm(answer) {
54    // Pretend it's hitting the network.
55    return new Promise((resolve, reject) => {
56    setTimeout(() => {
57    let shouldError = answer.toLowerCase() !== 'lima'
58    if (shouldError) {
59    reject(new Error('Good guess but a wrong answer. Try again!'));
60    } else {
61    resolve();
62    }
63    }, 1500);
64    });
65  }
```

Choosing the State Structure:

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs.

Principles for structuring state:

1. **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.
2. **Avoid contradictions in state.** When the state is structured in a way that several pieces of state may contradict and “disagree” with each other, you leave room for mistakes. Try to

avoid this.

3. **Avoid redundant state.** If you can calculate some information from the component's props or its existing state variables during rendering, you should not put that information into that component's state.
4. **Avoid duplication in state.** When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.
5. **Avoid deeply nested state.** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

Group related state:

We can use either one of them and it works

```
1   const[x, setX] =  useState(0);
2   const [y, setY] = useState(0);
```

Or:

```
1   const [position, setPosition] = useState({ x: 0, y: 0 });
```

Avoid contradictions in state

```
1   const isSending = status === 'sending';
2    const isSent = status === 'sent';
```

Avoid redundant state:

When I have a first name and last name , the full name can be calculated from the two previous variables so you should not create a state called full name.

```
1   const fullName = firstName + ' ' + lastName;
```

Avoid duplication in state

To update state directly on the screen

Avoid deeply nested state

Deleting a deeply nested place would involve copying its entire parent place chain which is very bad. Instead of a tree-like structure where each place has an array of its child places, you can have each place hold an array of its child place IDs. Then store a mapping from each place ID to the corresponding place.

```
import { useState } from 'react';
import { initialTravelPlan } from './places.js';

export default function TravelPlan() {
 const [plan, setPlan] = useState(initialTravelPlan);
 function handleComplete(parentId, childId) {
 const parent = plan[parentId];
 // Create a new version of the parent place
 // that doesn't include this child ID.
 const nextParent = {
 ...parent,
 childIds: parent.childIds
 .filter(id => id !== childId)
 };
 // Update the root state object...
 setPlan({
 ...plan,
 // ...so that it has the updated parent.
 [parentId]: nextParent
 });
 }

 const root = plan[0];
 const planetIds = root.childIds;
 return (
 <>
 <h2>Places to visit</h2>
 <ol>
 {planetIds.map(id => (
 <PlaceTree
 key={id}
 id={id}
 parentId={0}
 placesById={plan}
 onComplete={handleComplete}
```

```
    />
    ))}
    </ol>
    </>
    );
  }
  function PlaceTree({ id, parentId, placesById, onComplete }) {
    const place = placesById[id];
    const childIds = place.childIds;
    return (
    <li>
    {place.title}
    <button onClick={() => {
    onComplete(parentId, id);
    }}>
    Complete
    </button>
    {childIds.length > 0 &&
    <ol>
    {childIds.map(childId => (
    <PlaceTree
    key={childId}
    id={childId}
    parentId={id}
    placesById={placesById}
    onComplete={onComplete}
    />
    ))}
    </ol>
    }
    </li>
    );
  }
```

Sharing State Between Components:

The state of two components can always change together. The solution is to remove state from both of them and move it to their closest common parent then pass it down via props. **(called lifting state up).**

Remove state from the child components

Remove such lines:

```
const [isActive, setIsActive] = useState(false);
```

and add these:

```
function Panel({ title, children, isActive }) {
...
}
```

Pass hardcoded data from the component parent

Locate the closest common parent component of both of the child components

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

export default function Accordion() {
 return (
 <>
 <h2>Almaty, Kazakhstan</h2>
 <Panel title="About" isActive={true}>
 With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
 </Panel>
 <Panel title="Etymology" isActive={true}>
```

The name comes from алма, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild Malus sieversii is considered a likely candidate for the ancestor of the modern domestic apple.

```
 </Panel>
 </>
 );
}

function Panel({ title, children, isActive }) {
 return (
```

```
  <section className="panel">
  <h3>{title}</h3>
  {isActive ? (
  <p>{children}</p>
  ) : (
  <button onClick={() => setIsActive(true)}>
  Show
  </button>
  )}
  </section>
  );
 }
```

Add state to the common parent

Instead of a Boolean value , it could use a number as the index of the active Panel for the state variable

```
<>
  <Panel
  isActive={activeIndex === 0}
  onShow={() => setActiveIndex(0)}
  >
  ...
  </Panel>
  <Panel
  isActive={activeIndex === 1}
  onShow={() => setActiveIndex(1)}
  >
  ...
  </Panel>
</>
```

Preserving and Resetting State

State is isolated between components

State is tied to a position in the render tree

Each piece of state is holding with the correct component by where that components sits in the render tree.

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

export default function App() {
 const counter = <Counter />;
 return (
 <div>
 {counter}
 {counter}
 </div>
 );
}

function Counter() {
 const [score, setScore] = useState(0);
 const [hover, setHover] = useState(false);

 let className = 'counter';
 if (hover) {
 className += ' hover';
 }

 return (
 <div
 className={className}
 onPointerEnter={() => setHover(true)}
 onPointerLeave={() => setHover(false)}
 >
 <h1>{score}</h1>
 <button onClick={() => setScore(score + 1)}>
 Add one
 </button>
 </div>
 );
}
```

In this example you can remove a counter.

```
import { useState } from 'react';
export default function App() {
 const [showB, setShowB] = useState(true);
```

```
  return (
  <div>
  <Counter />
  {showB && <Counter />}
  <label>
  <input
  type="checkbox"
  checked={showB}
  onChange={e => {
  setShowB(e.target.checked)
  }}
  />
  Render the second counter
  </label>
  </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
  if (hover) {
  className += ' hover';
  }

  return (
  <div
  className={className}
  onPointerEnter={() => setHover(true)}
  onPointerLeave={() => setHover(false)}
  >
  <h1>{score}</h1>
  <button onClick={() => setScore(score + 1)}>
  Add one
  </button>
  </div>
  );
```

```
43  }
```

If the same component at the same position, it's the same counter even if there are 2 counters each one a style.
Different components at the same position reset state: the counter changes to p, the counter is deleted and then p is added.

Extracting State Logic into a Reducer

If components with many state updates, a reducer single function outside is a solution.

Consolidate state logic with a reducer

Having a task app that add, remove, edit and delete.
Each of those event handlers are called `setTasks`.
You can move the state logic into a single function outside your component called "**reducer**".

```
1   import { useReducer } from 'react';
2   import AddTask from './AddTask.js';
3   import TaskList from './TaskList.js';
4
5   export default function TaskApp() {
6    const [tasks, dispatch] = useReducer(tasksReducer,
    initialTasks);
7
8    function handleAddTask(text) {
9    dispatch({
10   type: 'added',
11   id: nextId++,
12   text: text,
13   });
14   }
15
16   function handleChangeTask(task) {
17   dispatch({
18   type: 'changed',
19   task: task,
20   });
21   }
22
23   function handleDeleteTask(taskId) {
```

```
  dispatch({
  type: 'deleted',
  id: taskId,
  });
  }

  return (
  <>
  <h1>Prague itinerary</h1>
  <AddTask onAddTask={handleAddTask} />
  <TaskList
  tasks={tasks}
  onChangeTask={handleChangeTask}
  onDeleteTask={handleDeleteTask}
  />
  </>
  );
}

function tasksReducer(tasks, action) {
  switch (action.type) {
  case 'added': {
  return [
  ...tasks,
  {
  id: action.id,
  text: action.text,
  done: false,
  },
  ];
  }
  case 'changed': {
  return tasks.map((t) => {
  if (t.id === action.task.id) {
  return action.task;
  } else {
  return t;
  }
  });
  }
```

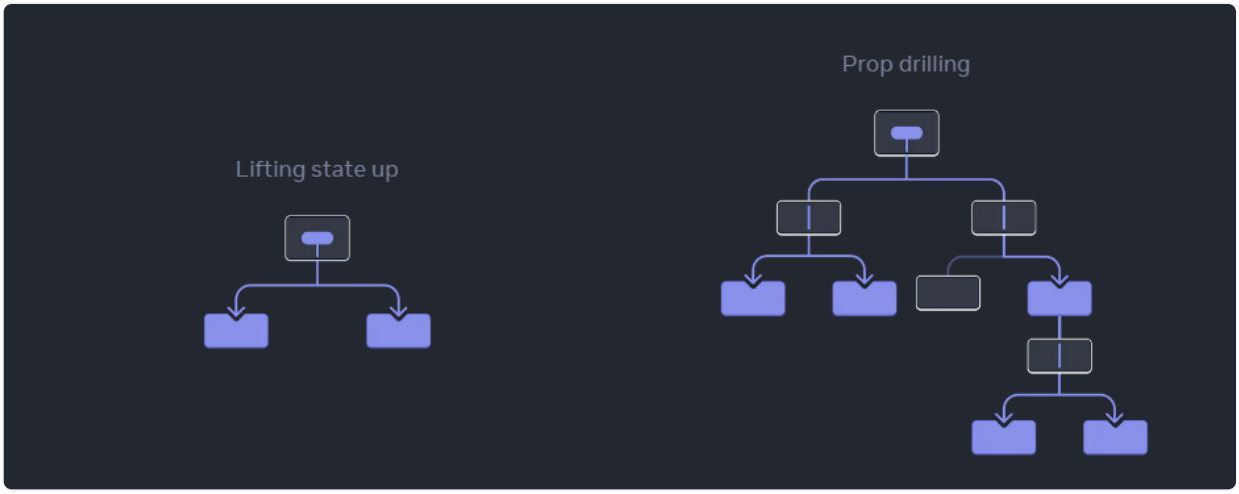
```
    case 'deleted': {
    return tasks.filter((t) => t.id !== action.id);
    }
    default: {
    throw Error('Unknown action: ' + action.type);
    }
    }
}

let nextId = 3;
const initialTasks = [
    {id: 0, text: 'Visit Kafka Museum', done: true},
    {id: 1, text: 'Watch a puppet show', done: false},
    {id: 2, text: 'Lennon Wall pic', done: false},
];
```

Passing Data Deeply with Context:

Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. *Context* lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

Passing props is a great way to explicitly pipe data through your UI tree to the components that use it. But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called “prop drilling”.



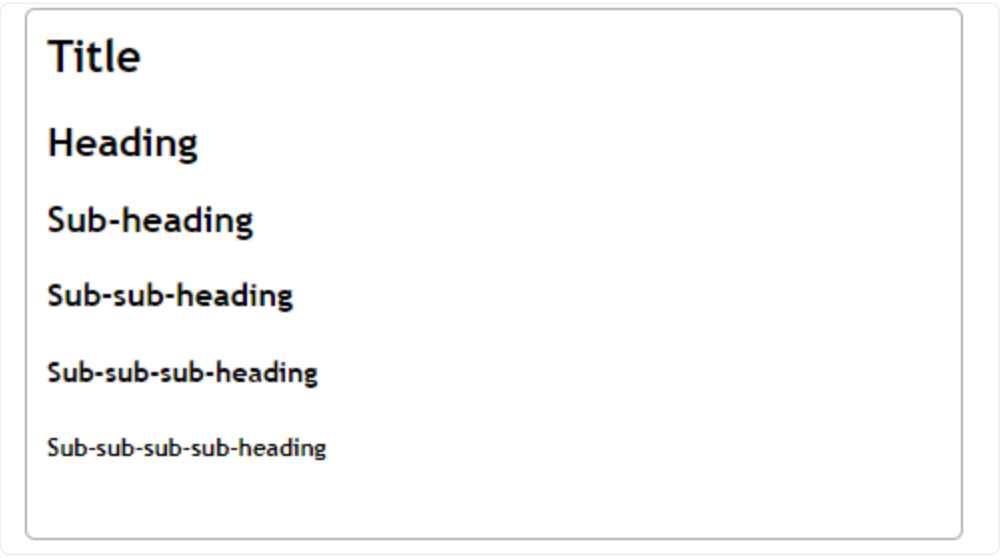
Context: an alternative to passing props

Context lets a parent component provide data to the entire tree below it. There are many uses for context. Here is one example. Consider this `Heading` component that accepts a `level` for its size:

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading level={1}>Title</Heading>
      <Heading level={2}>Heading</Heading>
      <Heading level={3}>Sub-heading</Heading>
      <Heading level={4}>Sub-sub-heading</Heading>
      <Heading level={5}>Sub-sub-sub-heading</Heading>
      <Heading level={6}>Sub-sub-sub-sub-heading</Heading>
    </Section>
  );
}
```

This code will give the following:

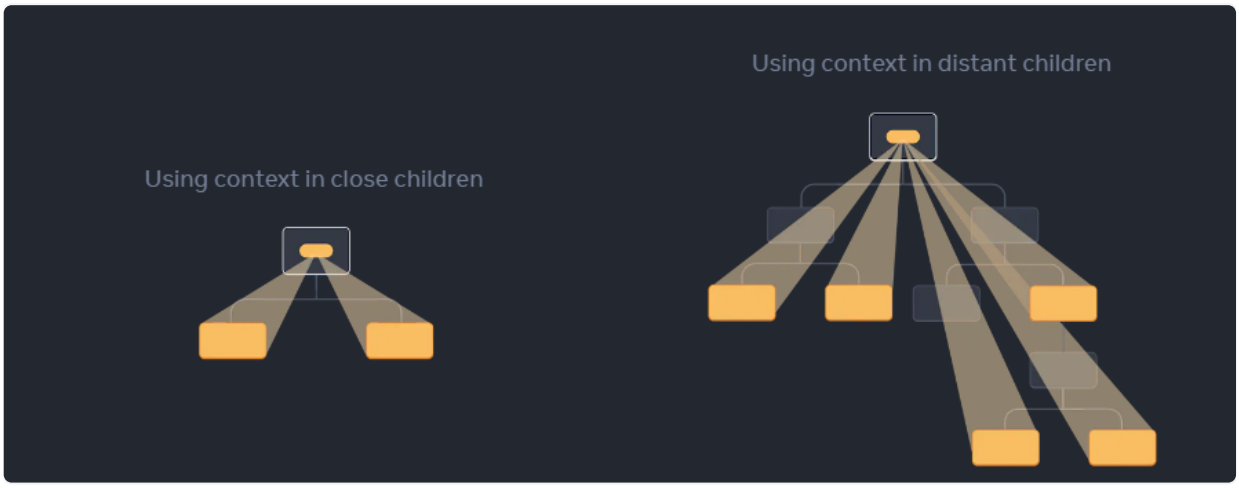


In order for the `<heading>` component to know the level of its closest `<Section>, t` hat would require some way for a child to "ask" for data from somewhere above in the tree.

You can’t do it with props alone. This is where context comes into play. You will do it in three steps:

1. **Create** a context. (You can call it `LevelContext`, since it’s for the heading level.)
2. **Use** that context from the component that needs the data. (`Heading` will use `LevelContext`.)
3. **Provide** that context from the component that specifies the data. (`Section` will provide `LevelContext`.)

Context lets a parent—even a distant one!—provide some data to the entire tree inside of it.



1- Create the context

First, you need to create the context. You’ll need to **export it from a file** so that your components can use it:

```
import Heading from './Heading.js';
import Section from './Section.js';
```

2- Use the Context:

```
export default function Heading({children}) {
const level= useContext (LevelContext);
}
```

`useContext` is a **Hook**. Just like `useState` and `useReducer`, you can only call a Hook immediately inside a React component (not inside loops or conditions). **`useContext` tells React that the `Heading` component wants to read the `LevelContext`.**

3-Provide the context

```
import {LevelContext} from './LevelContext.js';

export default function Section({level,children}){
 return(
  <section className ="section">
    <LevelContext.Providervalue={level}>
      {children}
    </LevelContext.Provider>
  </section>
  );
}
```

Scaling Up with Reducer and Context:

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components.

Combining a reducer with context:

In this example, the state is managed by a reducer. The reducer function contains all of the state update logic and is declared at the bottom of this file:

```
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
  }
```

```
    function handleChangeTask(task) {
      dispatch({
        type: 'changed',
        task: task
      });
    }

    function handleDeleteTask(taskId) {
      dispatch({
        type: 'deleted',
        id: taskId
      });
    }
    return (
      <>
        <h1>Day off in Kyoto</h1>
        <AddTask
          onAddTask={handleAddTask}
        />
        <TaskList
          tasks={tasks}
          onChangeTask={handleChangeTask}
          onDeleteTask={handleDeleteTask}
        />
      </>
    );
  }
  function tasksReducer(tasks, action) {
    switch (action.type) {
      case 'added': {
        return [...tasks, {
          id: action.id,
          text: action.text,
          done: false
        }];
      }
      case 'changed': {
        return tasks.map(t => {
          if (t.id === action.task.id) {
            return action.task;
```

```
          } else {
            return t;
          }
        });
      }
      case 'deleted': {
        return tasks.filter(t => t.id !== action.id);
      }
      default: {
        throw Error('Unknown action: ' + action.type);
      }
    }
  }
  let nextId = 3;
  const initialTasks = [
    { id: 0, text: 'Philosopher's Path', done: true },
    { id: 1, text: 'Visit the temple', done: false },
    { id: 2, text: 'Drink matcha', done: false }
  ];
```

The reducer helps keep the event handlers short and concise, but as the app get bigger, it will may run into another difficulty.

We can combine a reduces with context by:

1. **Create** the context.
2. **Put** state and dispatch into context.
3. **Use** context anywhere in the tree.

1- Create the Context:

```
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

To pass them down the tree, you will **create** two separate contexts:

- `TasksContext` provides the current list of tasks.
- `TasksDispatchContext` provides the function that lets components dispatch actions.
-

2- Put state and dispatch into context:

```
import { TasksContext, TasksDispatchContext } from
'./TasksContext.js';

export default function TaskApp() {
  const [tasks, dispatch] = useReducer(tasksReducer,
initialTasks);
  // ...
  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        ...
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}
```

3- Use context anywhere in the tree:

```
export default function TaskList() {
  const tasks = useContext(TasksContext);
  // ...
```

Escape Hatches:

Some of your components may need to control and synchronize with systems outside of React.

Referencing values with refs:

When you want a component to “remember” some information, but you don’t want that information to trigger new render, you can use a ref:

```
const ref = useRef(0);
```

Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not! You can access the current value of that ref through the `ref.current` property.

```
import { useRef } from 'react';

export default function Counter() {
  let ref = useRef(0);
  function handleClick() {
    ref.current = ref.current + 1;
    alert('You clicked ' + ref.current + ' times!');
  }
  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
}

```

This code will give an alert for every time the button is pressed.

Manipulating the DOM with refs:

React automatically updates the DOM to match your render output, so your components won't often need to manipulate it. But, sometimes we might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position.

How to write an effect:

1- Declare the Effect:

To declare an effect, we import the `useEffect` Hook from react:

```
import {useEffect} from 'react';
```

and then we call it at the top level of the component and enter some code inside the `Effect`:

```
function MyComponent() {
 useEffect(() => {
 // Code here will run after *every* render
 });
 return <div />;
```

```
}
```

2- Specify the Effect dependencies:

You can tell React to **skip unnecessarily re-running the Effect** by specifying an array of *dependencies* as the second argument to the `useEffect`

Synchronizing with Effects:

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Unlike event handlers, which let you handle particular events, Effects let you run some code after rendering. Use them to synchronize your component with a system outside of React.

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

function VideoPlayer({ src, isPlaying }) {
  const ref = useRef(null);

  useEffect(() => {
    if (isPlaying) {
      ref.current.play();
    } else {
      ref.current.pause();
    }
  }, [isPlaying]);

  return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
  const [isPlaying, setIsPlaying] = useState(false);
  return (
    <>
      <button onClick={() => setIsPlaying(!isPlaying)}>
        {isPlaying ? 'Pause' : 'Play'}
      </button>
      <VideoPlayer
```

```
          isPlaying={isPlaying}
          src="https://interactive-
    examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"    />
      </>
    );
  }
```

The above code will give a button and a picture. When pressed on the button the picture will move and start playing and when pressed again it will stop.

Add cleanup if needed:

React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed).

No need for cleanup in this case because calling zoomLevel twice will not make a difference. However, when calling showModal method of the built in <dialog> it will be a problem when we call it twice so it needs a clean up.

```
useEffect(() => {
 const dialog = dialogRef.current;

 dialog.showModal();

 return;
 () => dialog.close();
}, []);
```

Controlling non-React widgets:

Sometimes you need to add UI widgets that aren't written to react such as adding a map component.

```
useEffect(() => {
```

```
    const map = mapRef.current;
    map.setZoomLevel(zoomLevel);
  }, [zoomLevel]);
```

Subscribing to events:

If your Effect subscribes to something, the cleanup function should unsubscribe:

```
useEffect(() => {
  function handleScroll(e) {
    console.log(window.scrollX, window.scrollY);
  }
  window.addEventListener('scroll', handleScroll);
  return () => window.removeEventListener('scroll',
handleScroll);
}, []);
```

Triggering animations:

If your Effect animates something in, the cleanup function should reset the animation to the initial values

```
useEffect(() => {
  const node = ref.current;
  node.style.opacity = 1; // Trigger the animation
  return () => {
    node.style.opacity = 0; // Reset to the initial value
  };
}, []);
```

Fetching data:

If your Effect fetches something, the cleanup function should either abort the fetch or ignore its result:

```
useEffect(() => {
  let ignore = false;

  async function startFetching() {
```

```
      const json = await fetchTodos(userId);
      if (!ignore) {
        setTodos(json);
      }
    }
    startFetching();
    return () => {
      ignore = true;
    };
  }, [userId]);
```

You Might Not Need an Effect:

If there is no external system involved (for example, if you want to update a component's state when some props or state change), you shouldn't need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

How to remove unnecessary effects

- You don't need Effects to transform data for rendering.
- You don't need Effects to handle user events.
- Updating state based on props state

```
function Form() {
  const [firstName, setFirstName] = useState("Taylor");
  const [lastName, setLastName] = useState("Swift");
  //  Good: calculated during rendering
  const fullName = firstName + " " + lastName;
  // ...
}
```

Let it update directly while rendering, no need to create a state for it.

Resetting all state when a prop changes

```
export default function ProfilePage({ userId }) {
  return;
  <Profile userId={userId} key={userId} />;
}
function Profile({ userId }) {
  //  This and any other state below will reset on key change
automatically
```

```
  const [comment, setComment] = useState("");
  // ...
}
```

Sending a POST request:

```
function Form() {
  const [firstName, setFirstName] = useState("");

  const [lastName, setLastName] = useState("");
  //  Good: This logic runs because the component was displayed
  useEffect(() => {
    post("/analytics/event", {
      eventName: "visit_form",
    });
  }, []);
  function handleSubmit(e) {
    e.preventDefault();
    //  Good: Event-specific logic is in the event handler
    post("/api/register", {
      firstName,
      lastName,
    });
  }
  // ...
}
```

Fetching the data:

```
function SearchResults({ query }) {
  const [results, setResults] = useState([]);

  const [page, setPage] = useState(1);

  useEffect(() => {
    let ignore = false;

    fetchResults(query, page).then((json) => {
      if (!ignore) {
        setResults(json);
```

```
        }
      });

      return;
      () => {
        ignore = true;
      };
    }, [query, page]);

    function handleNextPageClick() {
      setPage(page + 1);
    }

    // ...
  }
```

Lifecycle of Reactive Effects:

Components may mount, update or unmount. Effect can start synchronizing and later to stop synchronizing it.

The lifecycle of on Effect:

- A component mounts when it’s added to the screen.
- A component updates when it receives new props or state, usually in response to an interaction.
-

A component unmounts when it’s removed from the screen. Thinking from the effect's perspective Let’s recap everything that’s happened from the ChatRoom component’s perspective:

- ChatRoom mounted with roomId set to "general"
- ChatRoom updated with roomId set to "travel"
- ChatRoom updated with roomId set to "music"
- ChatRoom unmounted

During each of these points in the component’s lifecycle, your Effect did different things:

- Your Effect connected to the "general" room
- Your Effect disconnected from the "general" room and connected to the "travel" room

- Your Effect disconnected from the "travel" room and connected to the "music" room
- Your Effect disconnected from the "music" room

```
useEffect(() => {
  // Your Effect connected to the room specified with roomId...
  const connection = createConnection(serverUrl, roomId);
  connection.connect();
  return;
  () => {
    // ...until it disconnected
    connection.disconnect();
  };
}, [roomId]);
```