Setup

- traditional Vite app
    - removed boilerplate
    - provided some assets (css, data)
        - just so we can focus on important stuff
    - removed , so it's less logs

Advanced Topics

- /tutorial directory
- work in the starter folder
- complete code in the final folder
- in order to work on topic import component from 'starter'
- in order to test final import component from 'final'
- setup challenges
- in the beginning examples with numbers and buttons 😃😃😃

```
import Starter from './tutorial/1-useState/starter/1-error-example';
import Final from './tutorial/1-useState/final/1-error-example';
function App() {
  return (
    <div className="container">
      <Starter />
      <Final />
    </div>
  );
}

export default App;
```

The Need For State

```
import Starter from './tutorial/01-useState/starter/01-error-
example.jsx';
```

- in App.jsx setup import and container div

  Setup Challenge :

- create count variable
- display value in the JSX
- add button and increase the value

- the reason for bug - we don't trigger re-render (reference next lecture)

```
const ErrorExample = () => {
  let count = 0;

  const handleClick = () => {
    count = count + 1;
    console.log(count);
    // preserve value between renders
    // trigger re-render
  };
  return (
    <div>
      <h2>{count}</h2>
      <button type="button" className="btn" onClick={handleClick}>
        increment
      </button>
    </div>
  );
};

export default ErrorExample;
```

useState Basics

```
import Starter from './tutorial/01-useState/starter/02-useState-
basics.jsx';
```

Javascript Nuggets - Destructuring (Array)

- useState hook
- returns an array with two elements: the current state value, and a function that we can use to update the state
- accepts default value as an argument
- state update triggers re-render

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

const UseStateBasics = () => {
  // console.log(useState());
  // console.log(useState('jo koy'));
  // const value = useState()[0];
  // const handler = useState()[1];
  // console.log(value, handler);

  const [count, setCount] = useState(0);
  const handleClick = () => {
    // console.log(count)
    setCount(count + 1);
    // be careful, we can set any value
    // setCount('pants');
  };
  return (
    <div>
      <h4>You clicked {count} times</h4>
      <button className="btn" onClick={handleClick}>
        Click me
      </button>
    </div>
  );
};

export default UseStateBasics;
```

Initial Render and Re-Renders

In a React application, the initial render is the first time that the component tree is rendered to the DOM. It happens when the application first loads, or when the root component is first rendered. This is also known as "mounting" the components.

Re-renders, on the other hand, happen when the component's state or props change, and the component needs to be updated in the DOM to reflect these changes. React uses a virtual DOM to optimize the process of updating the actual DOM, so that only the necessary changes are made.

There are a few ways that you can trigger a re-render in a React component:

- By changing the component's state or props. When the component's state or props change, React will re-render the component to reflect these changes.

- When the parent element re-renders, even if the component's state or props have not changed.

General Rules of Hooks

- starts with "use" (both -react and custom hooks)
- component must be uppercase
- invoke inside function/component body
- don't call hooks conditionally (cover later)
- set functions don't update state immediately (cover later)

useState with Array

```
import Starter from './tutorial/01-useState/starter/03-useState-
array.jsx';
```

Setup Challenge :

- import data
- setup a state value
    - people - default value equal to data
- display list(people) in the browser
- create two functions
    - one that removes single item from the list
    - one that clears entire list

1. render the list

```
import React from 'react';
import { data } from '../../../data';
const UseStateArray = () => {
  const [people, setPeople] = React.useState(data);

  return (
    <div>
      {people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
          </div>
        );
      })}
    </div>
  );
};

export default UseStateArray;
```

2. remove items

Javascript Nuggets - Filter and Find

```
import React from 'react';
import { data } from '../../../data';
const UseStateArray = () => {
  const [people, setPeople] = React.useState(data);

  const removeItem = (id) => {
    let newPeople = people.filter((person) => person.id !== id);
    setPeople(newPeople);
  };
  return (
    <div>
      {people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
            <button onClick={() => removeItem(id)}>remove</button>
          </div>
        );
      })}
      <button
        className="btn"
        style={{ marginTop: '2rem' }}
        onClick={() => setPeople([])}
      >
        clear items
      </button>
    </div>
  );
};

export default UseStateArray;
```

- should we update backroads app project?

useState with Object

```
import Starter from './tutorial/01-useState/starter/04-useState-
object.jsx';
```

Setup Challenge :

- setup three state values
  - name(string)
  - age(number)
  - hobby(string)
- render in the browser
- create a button
  - setup a function
    - update all three state values
- as a result once the user clicks the button,
  new person is displayed in the browser

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

```
const UseStateObject = () => {
  const [name, setName] = useState('peter');
  const [age, setAge] = useState(24);
  const [hobby, setHobby] = useState('read books');

  const displayPerson = () => {
    setName('john');
    setAge(28);
    setHobby('scream at the computer');
  };
  return (
    <>
      <h3>{name}</h3>
      <h3>{age}</h3>
      <h4>Enjoys To: {hobby}</h4>
      <button className="btn" onClick={displayPerson}>
        show john
      </button>
    </>
  );
};

export default UseStateObject;
```

Automatic Batching

In React, "batching" refers to the process of grouping multiple state updates into a single update. This can be useful in certain cases because it allows React to optimize the rendering of your components by minimizing the number of DOM updates that it has to perform.

By default, React uses a technique called "auto-batching" to group state updates that occur within the same event loop into a single update. This means that if you call the state update function multiple times in a short period of time, React will only perform a single re-render for all of the updates.

React 18 ensures that state updates invoked from any location will be batched by default. This will batch state updates, including native event handlers, asynchronous operations, timeouts, and intervals.

Switch to Object

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

const UseStateObject = () => {
  const [person, setPerson] = useState({
    name: 'peter',
    age: 24,
    hobby: 'read books',
  });

  const displayPerson = () => {
    setPerson({ name: 'john', age: 28, hobby: 'scream at the computer'
});
    // be careful, don't overwrite
    // setPerson('shakeAndBake');
    // setPerson({ name: 'susan' });
    // setPerson({ ...person, name: 'susan' });
  };
  return (
    <>
      <h3>{person.name}</h3>
      <h3>{person.age}</h3>
      <h4>Enjoys To: {person.hobby}</h4>
      <button className="btn" onClick={displayPerson}>
        show john
      </button>
    </>
  );
};

export default UseStateObject;
```

Set Function "Gotcha"

```
import Starter from './tutorial/01-useState/starter/05-useState-
gotcha.jsx';
```

Setup Challenge :

- setup a state value and the button
- add functionality to increase value by 1
- log a state value, right after setFunction

Keep in mind that the state update function setState does not immediately mutate the state. Instead, it schedules an update to the state and tells React that it needs to re-render the component. The actual state update will be performed as part of the next rendering cycle. Be mindful when you need to set state value based on a different state value.

trivial example

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

const UseStateGotcha = () => {
  const [value, setValue] = useState(0);

  const handleClick = () => {
    setValue(value + 1);
    //  be careful it's the old value
    console.log(value);
    //  so if you have any functionality
    // that relies on the latest value
    // it will be wrong !!!
  };
  return (
    <div>
      <h1>{value}</h1>
      <button className="btn" onClick={handleClick}>
        increase
      </button>
    </div>
  );
};

export default UseStateGotcha;
```

If you want to update the state immediately and synchronously, you can pass a function to setState that receives the previous state as an argument and returns the new state. For example:

```
setState((prevState) => {
  return { ...prevState, value: newValue };
});
```

This can be useful if you need to update the state based on the previous state, or if you need to update the state synchronously.

```
const handleClick = () => {
  setValue((currentState) => {
    // must return otherwise undefined
    // below is the latest/current state value
    const newState = currentState + 1;
    return newState;
  });
};
```

- setTimeout Example

```
const handleClick = () => {
  // setTimeout(() => {
  // console.log('clicked the button');
  //   setValue(value + 1);
  // }, 3000);
  setTimeout(() => {
    console.log('clicked the button');
    setValue((currentState) => {
      return currentState + 1;
    });
  }, 3000);
};
```

- as an example refactor code in /tutorial/01-useState/03-useState-array
- should we use functional update approach for everything?

Code Example

```
import Starter from './tutorial/02-useEffect/starter/01-code-
example.jsx';
```

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

const ComponentExample = () => {
  const [value, setValue] = useState(0);
  const sayHello = () => {
    console.log('hello there');
    // be careful
    // setValue(value + 1);
  };
  sayHello();
  return (
    <div>
      <h1>value : {value}</h1>
      <button className="btn" onClick={() => setValue(value + 1)}>
        click me
      </button>
    </div>
  );
};
export default ComponentExample;
```

- the problem starts when we update the state

```
const [value, setValue] = useState(0);

const sayHello = () => {
  console.log('hello there');
  // be careful, you will have infinite loop
  setValue(value + 1);
};
sayHello();
```

- initial render - setup state value and invoke sayHello
- in the sayHello update state, trigger re-render
- re-render - setup state value and invoke sayHello
- in the sayHello update state, trigger re-render
- repeat
- repeat
- repeat
  ....
- but what about fetching data?

useEffect Basics

```
import Starter from './tutorial/02-useEffect/starter/02-useEffect-
basics.jsx';
```

useEffect is a hook in React that allows you to perform side effects in function components.There is no need for urban dictionary - basically any work outside of the component. Some examples of side effects are: subscriptions, fetching data, directly updating the DOM, event listeners, timers, etc.

- useEffect hook
- accepts two arguments (second optional)
- first argument - callback function
- second argument - dependency array
- by default runs on each render (initial and re-render)
- cb can't return promise (so can't make it async)
- if dependency array empty [] runs only on initial render

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

const UseEffectBasics = () => {
  const [value, setValue] = useState(0);
  const sayHello = () => {
    console.log('hello there');
  };

  sayHello();

  // useEffect(() => {
  //   console.log('hello from useEffect');
  // });
  useEffect(() => {
    console.log('hello from useEffect');
  }, []);
  return (
    <div>
      <h1>value : {value}</h1>
      <button className="btn" onClick={() => setValue(value + 1)}>
        click me
      </button>
    </div>
  );
};
export default UseEffectBasics;
```

Multiple Effects

```
import Starter from './tutorial/02-useEffect/starter/03-multiple-
effects.jsx';
```

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

const MultipleEffects = () => {
  const [value, setValue] = useState(0);
  const [secondValue, setSecondValue] = useState(0);

  useEffect(() => {
    console.log('hello from first useEffect');
  }, [value]);

  useEffect(() => {
    console.log('hello from second useEffect');
  }, [secondValue]);
  return (
    <div>
      <h1>value : {value}</h1>
      <button className="btn" onClick={() => setValue(value + 1)}>
        value
      </button>
      <h1>second value : {secondValue}</h1>
      <button className="btn" onClick={() => setSecondValue(secondValue
+ 1)}>
        second value
      </button>
    </div>
  );
};
export default MultipleEffects;
```

Fetch Data

```
import Starter from './tutorial/02-useEffect/starter/04-fetch-data.jsx';
```

Javascript Nuggets - Fetch API

- later in the course we will use axios

Setup Challenge :

- import useState and useEffect
- setup state value
  - users - default value []
- setup useEffect
- MAKE SURE IT RUNS ONLY ON INITIAL RENDER
- in the cb, create a function which performs fetch functionality
  - use url I provided in the starter file
  - you can use .then or async
  - set users equal to result
  - iterate over the list and display image, user name and link
- DON'T WORRY ABOUT CSS, MOST IMPORTANT LOGIC !!!

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

const url = 'https://api.github.com/users';

const FetchData = () => {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    // you can also setup function outside
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        const users = await response.json();
        setUsers(users);
      } catch (error) {
        console.log(error);
      }
    };
    fetchData();
  }, []);
  return (
    <section>
      <h3>github users</h3>
      <ul className="users">
```

```
        {users.map((user) => {
          const { id, login, avatar_url, html_url } = user;
          return (
            <li key={id}>
              <img src={avatar_url} alt={login} />
              <div>
                <h5>{login}</h5>
                <a href={html_url}>profile</a>
              </div>
            </li>
          );
        })}
      </ul>
    </section>
  );
};
export default FetchData;
```

Cleanup Function

```
import Starter from './tutorial/02-useEffect/starter/05-cleanup-
function.jsx';
```

Will Cover After 03-conditional-rendering

- Setup Challenge :
- create state value
- in jsx return button which toggles state value
- based on condition return second component (simple return)
- inside second component create useEffect and run it only on initial render

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

const CleanupFunction = () => {
  const [toggle, setToggle] = useState(false);
  return (
    <div>
      <button className="btn" onClick={() => setToggle(!toggle)}>
        toggle component
      </button>
      {toggle && <RandomComponent />}
    </div>
  );
};
```

```
const RandomComponent = () => {
  useEffect(() => {
    console.log('hmm, this is interesting');
  }, []);
  return <h1>hello there</h1>;
};
export default CleanupFunction;
```

Vanilla JS

```
const intID = setInterval(() => {
  console.log('hello from interval');
}, 1000);
clearInterval(intID);
```

```
const someFunc = () => {
  // some logic here
};
window.addEventListener('scroll', someFunc);
window.removeEventListener('scroll', someFunc);
```

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

const CleanupFunction = () => {
  const [toggle, setToggle] = useState(false);
  return (
    <div>
      <button className="btn" onClick={() => setToggle(!toggle)}>
        toggle component
      </button>
      {toggle && <RandomComponent />}
    </div>
  );
};
const RandomComponent = () => {
  useEffect(() => {
    // console.log('hmm, this is interesting');
    const intID = setInterval(() => {
      console.log('hello from interval');
    }, 1000);
    // does not stop, keeps going
    // every time we render component new interval gets created
    return () => clearInterval(intID);
  }, []);
  return <h1>hello there</h1>;
};
export default CleanupFunction;
```

```
useEffect(() => {
  // console.log('hmm, this is interesting');
  const someFunc = () => {
    // some logic here
  };
  window.addEventListener('scroll', someFunc);
  return () => window.removeEventListener('scroll', someFunc);
}, []);
```

You Might Not Need an Effect

You Might Not Need an Effect

- will still utilize useEffect
- there is still plenty of code using useEffect
- fetching data
  replaced by libraries - react query, rtk query, swr or next.js

```
import { useHook } from 'library';

function Example() {
  const { data, error, isLoading } = useHook('url', fetcher);

  if (error) return <div>failed to load</div>;
  if (isLoading) return <div>loading...</div>;
  return <div>hello {data.name}!</div>;
}
```

- rest of them by refactoring code

Multiple Returns - Basics

```
import Starter from './tutorial/03-conditional-rendering/starter/01-
multiple-returns-basics.jsx';
```

Vanilla JS

```
const sayHello = (name) => {
  if (name) {
    return `Hello, ${name}`;
    // exit the function, skip rest of the code
  }
  // so if name provided, won't get to this line
  return 'Hello, there';
};

const firstResp = sayHello('john');
console.log(firstResp); // Hello, john
const secondResp = sayHello();
console.log(secondResp); // Hello, there
```

- if no explicit return by default function returns 'undefined'

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

const MultipleReturnsBasics = () => {
  // while fetching data
  // convention with boolean values "isSomething"
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setTimeout(() => {
      // done fetching data
      setIsLoading(false);
    }, 3000);
  }, []);

  // can return entire app
  if (isLoading) {
    return <h2>Loading...</h2>;
  }

  return <h2>My App</h2>;
};
export default MultipleReturnsBasics;
```

Multiple Returns - Fetch Data

```
import Starter from './tutorial/03-conditional-rendering/starter/02-
multiple-returns-fetch-data.jsx';
```

Setup Challenge :

- practice on setting up state values and data fetching
- create state variable
  - user - default value null
- fetch data from the url (for now just log result)
- if you see user object in the console, continue with the videos

```
import { useEffect, useState } from 'react';
const url = 'https://api.github.com/users/QuincyLarson';

const MultipleReturnsFetchData = () => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const resp = await fetch(url);
        const user = await resp.json();
        console.log(user);
      } catch (error) {
        // fetch only cares about network errors
        // will work with axios
        console.log(error);
      }
    };
    fetchUser();
  }, []);

  return <h2>Fetch Example</h2>;
};
export default MultipleReturnsFetchData;
```

Data Fetching :

- usually three options
  - loading - waiting for data to arrive (display loading state)
  - error - there was an error (display error message)
  - success - received data (display data)

```
import { useEffect, useState } from 'react';
const url = 'https://api.github.com/users/QuincyLarson';

const MultipleReturnsFetchData = () => {
  // convention to setup booleans with isSomething
  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);
  const [user, setUser] = useState(null);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const resp = await fetch(url);
        const user = await resp.json();
        // console.log(user);
        setUser(user);
      } catch (error) {
        setIsError(true);
        console.log(error);
      }
      // hide loading
      setIsLoading(false);
    };
    fetchUser();
  }, []);
  // order matters
  // don't place user JSX before loading or error
  if (isLoading) {
    return <h2>Loading...</h2>;
  }
  if (isError) {
    return <h2>There was an error...</h2>;
  }
  return (
    <div>
      <img
        style={{ width: '150px', borderRadius: '25px' }}
        src={user.avatar_url}
        alt={user.name}
      />
      <h2>{user.name}</h2>
      <h4>works at {user.company}</h4>
      <p>{user.bio}</p>
    </div>
  );
};
export default MultipleReturnsFetchData;
```

Fetch Errors "Gotcha" (optional)

```
import Starter from './tutorial/03-conditional-rendering/starter/02-
multiple-returns-fetch-data.jsx';
```

Unlike for example Axios, by default, the fetch() API does not consider HTTP status codes in the 4xx or 5xx range to be errors. Instead, it considers these status codes to be indicative of a successful request,

```
try {
const resp = await fetch(url);
```

```
  // console.log(resp);
  if (!resp.ok) {
    setIsError(true);
    setIsLoading(false);
    return;
  }

  const user = await resp.json();
  setUser(user);

  }
```

Order Matters - Setup

```
import Starter from './tutorial/03-conditional-rendering/starter/02-
multiple-returns-fetch-data.jsx';
```

Please don't dismiss this topic. A lot of questions in course Q&A.

Challenge :

- destructure properties and remove user from JSX
- you might or might not encounter the bug

```
return (
  <div>
    <img
      style={{ width: '100px', borderRadius: '25px' }}
      src={avatar_url}
      alt={name}
    />
    <h2>{name}</h2>
    <h4>works at {company}</h4>
    <p>{bio}</p>
  </div>
);
```

Order Matters - Solution

- before returns

```
const [user, setUser] = useState(null);
console.log(user); // still null
```

```
// we can't pull out properties from null
const { avatar_url, name, company, bio } = user;
```

- after returns

```
console.log(user); // user object;
const { avatar_url, name, company, bio } = user;
```

```
return (
  <div>
    <img
      style={{ width: '100px', borderRadius: '25px' }}
      src={avatar_url}
      alt={name}
    />
    <h2>{name}</h2>
    <h4>works at {company}</h4>
    <p>{bio}</p>
  </div>
);
```

Vanilla JS

```
const someObject = {
  name: 'jo koy',
};
// this is cool
someObject.name; // returns 'jo koy'
someObject.propertyThatDoesNotExist; // returns undefined

// not cool at all, javascript will scream, yell and complain
const randomValue = null;
randomValue.name;

// this is ok
const randomList = [];
console.log(randomList[0]); // returns undefined

// not cool at all, javascript will scream, yell and complain
console.log(randomList[0].name);
```

Fetch Function Location

```
const fetchData = async () => {
  // fetch data
};

useEffect(() => {
  fetchData();
}, []);
```

- DON'T ADD fetchData to dependency array !!!
- IT WILL TRIGGER INFINITE LOOP !!!

DON'T CALL HOOKS CONDITIONALLY

```
import Starter from './tutorial/03-conditional-rendering/starter/03-
hooks-rule.jsx';
```

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

const Example = () => {
  const [condition, setCondition] = useState(true);
  if (condition) {
    // won't work
    const [state, setState] = useState(false);
  }

  if (condition) {
    return <h2>Hello There</h2>;
  }
  // this will also fail
  useEffect(() => {
    console.log('hello there');
  }, []);
  return <h2>example</h2>;
};

export default Example;
```

Truthy and Falsy Values (optional)

Vanilla JS

In JavaScript, a value is considered "truthy" if it is evaluated to true when used in a boolean context. A value is considered "falsy" if it is evaluated to false when used in a boolean context.

Here is a list of values that are considered falsy in JavaScript:

false
0 (zero)
"" (empty string)
null
undefined
NaN (Not a Number)
All other values, including objects and arrays, are considered truthy.

For example:

```
const x = 'Hello';
const y = '';
const z = 0;

if (x) {
  console.log('x is truthy');
}

if (y) {
  console.log('y is truthy');
} else {
  console.log('y is falsy');
}

if (z) {
  console.log('z is truthy');
} else {
  console.log('z is falsy');
}

// Output:
// "x is truthy"
// "y is falsy"
// "z is falsy"
```

In this example, the variable x is a non-empty string, which is considered truthy, so the first if statement is executed. The variable y is an empty string, which is considered falsy, so the else block of the second if statement is executed. The variable z is the number 0, which is considered falsy, so the else block of the third if statement is executed.

Short Circuit Evaluation (optional)

Vanilla JS

In JavaScript, short-circuit evaluation is a technique that allows you to use logical operators (such as && and ||) to perform conditional evaluations in a concise way.

The && operator (logical AND) returns the first operand if it is "falsy", or the second operand if the first operand is "truthy".

For example:

```
const x = 0;
const y = 1;

console.log(x && y); // Output: 0 (the first operand is falsy, so it is
returned)
console.log(y && x); // Output: 0 (the second operand is falsy, so it is
returned)
```

The || operator (logical OR) returns the first operand if it is "truthy", or the second operand if the first operand is "falsy".

For example:

```
const x = 0;
const y = 1;

console.log(x || y); // Output: 1 (the first operand is falsy, so the
second operand is returned)
console.log(y || x); // Output: 1 (the first operand is truthy, so it is
returned)
```

Short-circuit evaluation can be useful in cases where you want to perform a certain action only if a certain condition is met, or you want to return a default value if a certain condition is not met.

For example:

```
function displayName(name) {
  return name || 'Anonymous';
}

console.log(displayName('Pizza')); // Output: "Pizza"
console.log(displayName()); // Output: "Anonymous"
```

In this example, the displayName() function returns the name property of the user object if it exists, or "Anonymous" if the name property is not present. This is done using the || operator and short-circuit evaluation.

Short Circuit Evaluation React - Basics

```
import Starter from './tutorial/03-conditional-rendering/starter/04-
short-circuit-overview.jsx';
```

Setup Challenge :

- create two state values
- one "falsy" and second "truthy"
- setup both conditions for each operator in JSX - hint {}
    - || OR
    - && AND

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

const ShortCircuitOverview = () => {
  // falsy
  const [text, setText] = useState('');
  // truthy
  const [name, setName] = useState('susan');

  const codeExample = text || 'hello world';

  // can't use if statements
  return (
    <div>
      {/* {if(someCondition){"won't work"}} */}

      <h4>Falsy OR : {text || 'hello world'}</h4>
      <h4>Falsy AND {text && 'hello world'}</h4>
      <h4>Truthy OR {name || 'hello world'}</h4>
      <h4>Truthy AND {name && 'hello world'}</h4>
      {codeExample}
    </div>
  );
};
export default ShortCircuitOverview;
```

Short Circuit Evaluation in React - Common Approaches

```
import Starter from './tutorial/03-conditional-rendering/starter/05-
short-circuit-examples.jsx';
```

Vanilla JS (Optional)
The ! operator is a logical operator in JavaScript that negates a boolean value. It is equivalent to the not operator in other programming languages.

For example:

```
let isTrue = true;
let isFalse = false;
```

```
console.log(!isTrue); // outputs: false
console.log(!isFalse); // outputs: true
```

You can use the ! operator to test if a value is not truthy or falsy:

```
let val = 0;
if (!val) {
  console.log('val is falsy');
}
```

You can also use the ! operator to convert a value to a boolean and negate it:

```
let val = 'hello';
let bool = !val; // bool is now false

val = '';
bool = !val; // bool is now true
```

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

const ShortCircuitOverview = () => {
  // falsy
  const [text, setText] = useState('');
  // truthy
  const [name, setName] = useState('susan');
  const [user, setUser] = useState({ name: 'john' });
  const [isEditing, setIsEditing] = useState(false);

  // can't use if statements
  return (
    <div>
      <h2>{text || 'default value'}</h2>
      {text && (
        <div>
          <h2> whatever return</h2>
          <h2>{name}</h2>
        </div>
      )}
      {/* more info below */}
      {!text && (
        <div>
          <h2> whatever return</h2>
          <h2>{name}</h2>
        </div>
      )}
```

```
        {user && <SomeComponent name={user.name} />}
        <h2 style={{ margin: '1rem 0' }}>Ternary Operator</h2>
        <button className="btn">{isEditing ? 'edit' : 'add'}</button>
        {user ? (
          <div>
            <h4>hello there user {user.name}</h4>
          </div>
        ) : (
          <div>
            <h2>please login</h2>
          </div>
        )}
      </div>
    );
  };

  const SomeComponent = ({ name }) => {
    return (
      <div>
        <h4>hello there, {name}</h4>
        <button className="btn">log out</button>
      </div>
    );
  };
  export default ShortCircuitEvaluation;
```

Ternary Operator

Vanilla JS

In JavaScript, the ternary operator is a way to concisely express a simple conditional statement. It is often called the "conditional operator" or the "ternary conditional operator".

Here is the basic syntax for using the ternary operator:

```
condition ? expression1 : expression2;
```

If condition is truthy, the operator will return expression1. If condition is falsy, it will return expression2.

Jobster Example

Jobster

Toggle Challenge

```
import Starter from './tutorial/03-conditional-rendering/starter/06-toggle-challenge.jsx';
```

- create state value (boolean)
- return a button and a component/element
- when user clicks the button
  - toggle state value
  - conditionally render component/element

Initial Setup

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

const ToggleChallenge = () => {
  const [showAlert, setShowAlert] = useState(false);

  const toggleAlert = () => {
    if (showAlert) {
      setShowAlert(false);
      return;
    }
    setShowAlert(true);
  };

  return (
    <div>
      <button className="btn" onClick={toggleAlert}>
        toggle alert
      </button>
      {showAlert && <Alert />}
    </div>
  );
};

const Alert = () => {
  return <div className="alert alert-danger">hello world</div>;
};
export default ToggleChallenge;
```

Improvements

```
<button className='btn' onClick={() => setShowAlert(!showAlert)}>
```

User Challenge

```
import Starter from './tutorial/03-conditional-rendering/starter/07-
user-challenge.jsx';
```

- create state value
  - user - default value null
- create two functions
  - login - set's user equal to object with name property
  - logout - set's user equal to null
- in jsx use ? to display two different setups
- h4 with "hello there, user name" and logout button
- h4 with "please login " and login button

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

const UserChallenge = () => {
  const [user, setUser] = useState(null);

  const login = () => {
    // normally connect to db or api
    setUser({ name: 'vegan food truck' });
  };
  const logout = () => {
    setUser(null);
  };

  return (
    <div>
      {user ? (
        <div>
          <h4>hello there, {user.name}</h4>
          <button className="btn" onClick={logout}>
            logout
          </button>
        </div>
      ) : (
        <div>
          <h4>Please Login</h4>
          <button className="btn" onClick={login}>
            login
          </button>
        </div>
      )}
    </div>
  );
};

export default UserChallenge;
```

Project Structure - Default Export

/tutorial/04-project-structure/starter

There are more options

Normally somewhere in the src

/components/componentName.jsx
/screens/componentName.jsx

- create navbar folder
    - setup Navbar.jsx (component)
    - Navbar.css (styles)
- import in App.jsx

import Final from 'pathToFolder/Navbar/Navbar'

- first solution rename to index.jsx(entry point)

Works but eventually too many index tabs 😃😃😃

- rename back to Navbar.jsx
- create index.jsx

```
export { default } from './Navbar';
```

Project Structure - Named Exports

/tutorial/04-project-structure/starter

- only makes sense if you have quite a few files
- create Pages directory
- setup two components Home.jsx and About.jsx
- import both in the App.jxs

import Home from 'pathToFolder/Pages/Home';
import About from 'pathToFolder/Pages/About';

A lot of work/lines of code

- create index.jsx

```
import Home from './Home';
import About from './About';
```

```
export { Home, About };
```

in App.jsx

import {Home, About} from 'pathToFolder/Pages

Project Structure - Export Group

/tutorial/04-project-structure/starter

- create Example directory
- setup two components (setup simple returns) and index.jsx file
- in index.jsx setup return and render both components (import)
- import/render index.jsx in App.jsx

Project Structure - Extra Extensions

- code spell checker - works well with code and documents.
- glean - easy extract JSX into a new component

Leverage Javascript

Javascript Nuggets -Optional Chaining

/tutorial/05-leverage-javascript/starter

Setup Challenge

- take a look at the people in array in data.js
- create List.jsx component
- in List.jsx import and iterate over people (data)
- for now just render name
- once you have list setup separate Person.jsx component
  - try glean extension
- in Person render
  - name, nickName, image

Yes, there will be a bug.

```
import { people } from '../../../data';

const List = () => {
  return (
    <div>
      {people.map((person) => {
        return <div>{person.name}</div>;
      })}
    </div>
  );
};
export default List;
```

List.jsx

```
import { people } from '../../../data';
import Person from './Person';
const List = () => {
  return (
    <div>
      {people.map((person) => {
        return <Person key={person.name} {...person} />;
      })}
    </div>
  );
};
export default List;
```

Person.jsx

```
import React from 'react';
import avatar from '../../../assets/default-avatar.svg';

export function Person({ name, nickName = 'shakeAndBake', images }) {
  // before optional chaining

  // const img =
  //   (images && images[0] && images[0].small && images[0].small.url)
|| avatar;
  // Combining with the nullish coalescing operator ??
  // const img = images?.[0]?.small?.url ?? avatar;
  // ?? vs || - please utilize the search engine

  const img = images?.[0]?.small?.url || avatar;

  return (
    <div>
      <img src={img} alt={name} style={{ width: '50px' }} />
      <h4>{name} </h4>
      <p>Nickname : {nickName}</p>
    </div>
  );
}
```

Default Values - Vanilla JS (Optional)

In JavaScript, when defining a function, you can specify default values for its parameters. This means that if a caller of the function does not provide a value for a particular parameter, the default value will be used instead. Default parameters are defined by assigning a value to the parameter in the function definition.

For example, consider the following function, which takes two parameters, x and y, and returns their sum:

```
function add(x, y) {
  return x + y;
}
```

If we call this function with only one argument, it will return NaN because y is undefined.

We can set default values for x,y as:

```
function add(x = 0, y = 0) {
  return x + y;
}
```

Now, if we call add(3), the function will return 3, because the default value of 0 is used for the y parameter.

Optional Chaining - Vanilla JS (Optional)

n JavaScript, the optional chaining operator (?.) is a new feature that allows you to access properties of an object without worrying about whether the object or the property is null or undefined. It's a shorthand for a common pattern of checking for null or undefined before accessing an object's property.

For example, consider the following code, which accesses the firstName property of an object:

```
const person = { name: { first: 'John', last: 'Doe' } };
console.log(person.name.first);
```

If the name property is null or undefined, this code will throw an error. To prevent this, we can use the optional chaining operator:

```
console.log(person?.name?.first);
```

Now, if the person.name is null or undefined, this code will simply return undefined instead of throwing an error. This make the code more robust and readable.

Controlled Inputs - Setup

```
import Starter from './tutorial/06-forms/starter/01-controlled-
inputs.jsx';
```

Setup (for all form videos)

```
const ControlledInputs = () => {
  return (
    <form className="form">
      <h4>controlled inputs</h4>
      <div className="form-row">
        <label htmlFor="name" className="form-label">
          name
        </label>
        <input type="text" className="form-input" id="name" />
      </div>
      <div className="form-row">
        <label htmlFor="email" className="form-label">
          Email
        </label>
        <input type="email" className="form-input" id="email" />
      </div>
      <button type="submit" className="btn btn-block">
        submit
      </button>
    </form>
  );
};
export default ControlledInputs;
```

Controlled Inputs - Complete

```
import Starter from './tutorial/06-forms/starter/01-controlled-
inputs.jsx';
```

- setup state values
- add value and onChange to each input
- setup onSubmit

```
import { useState } from 'react';
const ControlledInputs = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
```

```
    // const handleChange = (e) => {
    //   // for now we won't use it
    //   const name = e.target.name;
    //   const value = e.target.value;
    // };

    const handleSubmit = (e) => {
      e.preventDefault();
      // do something
      console.log(name, email);
    };
    return (
      <form className="form" onSubmit={handleSubmit}>
        <h4>controlled inputs</h4>
        <div className="form-row">
          <label htmlFor="name" className="form-label">
            name
          </label>
          <input
            type="text"
            className="form-input"
            value={name}
            onChange={(e) => setName(e.target.value)}
            id="name"
          />
        </div>
        <div className="form-row">
          <label htmlFor="email" className="form-label">
            Email
          </label>
          <input
            type="email"
            className="form-input"
            id="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
        </div>
        <button type="submit" className="btn btn-block">
          submit
        </button>
      </form>
    );
  };
  export default ControlledInputs;
```

User Challenge

```
  import Starter from './tutorial/06-forms/starter/02-user-challenge.jsx';
```

- setup controlled input (name input)
- setup onSubmit (for now just placeholder)
- import data array (first array) from data
- create another state value (data as default)
- iterate over and display right after form (h4)
- when user submits the form add new person to the list
- Extra Challenge
    - add button and setup functionality to remove user

```
import { useState } from 'react';
import { data } from '../../../data';
const UserChallenge = () => {
  const [name, setName] = useState('');
  const [users, setUsers] = useState(data);

  const handleSubmit = (e) => {
    e.preventDefault();
    // do something
    console.log(name);
    // if no value, do nothing
    if (!name) return;
    // if value, setup new user and add to current users
    const fakeId = Date.now();
    console.log(fakeId);
    // const newUser = { id: fakeId, name: name };
    const newUser = { id: fakeId, name };
    const updatedUsers = [...users, newUser];
    setUsers(updatedUsers);
    // set back to empty
    setName('');
  };

  const removeUser = (id) => {
    const updatedUsers = users.filter((person) => person.id !== id);
    setUsers(updatedUsers);
  };
  return (
    <div>
      <form className="form" onSubmit={handleSubmit}>
        <h4>Add User</h4>
        <div className="form-row">
          <label htmlFor="name" className="form-label">
            name
          </label>
          <input
            type="text"
            className="form-input"
            value={name}
            onChange={(e) => setName(e.target.value)}
            id="name"
          />
        </div>

        <button type="submit" className="btn btn-block">
          submit
        </button>
      </form>
      {/* render users */}
      <h2>users</h2>

      {users.map((user) => {
        return (
          <div key={user.id}>
            <h4>{user.name}</h4>
            <button onClick={() => removeUser(user.id)} className="btn">
              remove
            </button>
          </div>
        );
      })}
    </div>
  );
};
export default UserChallenge;
```

Multiple Inputs

```
import Starter from './tutorial/06-forms/starter/03-multiple-
inputs.jsx';
```

Javascript Nuggets - Dynamic Object Keys

- inputs must have name attribute

```
import { useState } from 'react';
const MultipleInputs = () => {
  const [user, setUser] = useState({
    name: '',
    email: '',
    password: '',
  });
```

```
    const handleChange = (e) => {
      setUser({ ...user, [e.target.name]: e.target.value });
    };

    const handleSubmit = (e) => {
      e.preventDefault();
      console.log(user);
    };
    return (
      <div>
        <form className="form" onSubmit={handleSubmit}>
          <h4>Multiple Inputs</h4>
          {/* name */}
          <div className="form-row">
            <label htmlFor="name" className="form-label">
              name
            </label>
            <input
              type="text"
              className="form-input"
              id="name"
              name="name"
              value={user.name}
              onChange={handleChange}
            />
          </div>
          {/* email */}
          <div className="form-row">
            <label htmlFor="email" className="form-label">
              Email
            </label>
            <input
              type="email"
              className="form-input"
              id="email"
              name="email"
              value={user.email}
              onChange={handleChange}
            />
          </div>
          {/* password */}
          <div className="form-row">
            <label htmlFor="password" className="form-label">
              Password
            </label>
            <input
              type="password"
              className="form-input"
              id="password"
              name="password"
              value={user.password}
              onChange={handleChange}
            />
          </div>

          <button type="submit" className="btn btn-block">
            submit
          </button>
        </form>
      </div>
    );
  };
  export default MultipleInputs;
```

Other Inputs

```
import Starter from './tutorial/06-forms/starter/04-other-inputs.jsx';
```

```
import { useState } from 'react';
const frameworks = ['react', 'angular', 'vue', 'svelte'];
const OtherInputs = () => {
  const [shipping, setShipping] = useState(false);
  const [framework, setFramework] = useState('react');

  const handleShipping = (e) => {
    console.log(e.target.checked);
    setShipping(e.target.checked);
  };
  const handleFramework = (e) => {
    setFramework(e.target.value);
  };
  return (
    <div>
      <form className="form">
        <h4>Other Inputs</h4>
        {/* name */}
        <div className="form-row" style={{ textAlign: 'left' }}>
          <input
            type="checkbox"
            checked={shipping}
            id="shipping"
            name="shipping"
            onChange={handleShipping}
          />
          <label htmlFor="shipping"> Free Shipping </label>
        </div>
        <div className="form-row" style={{ textAlign: 'left' }}>
          <label htmlFor="framework" className="form-label">
            Framework
          </label>
          <select
```

```
            name="framework"
            id="framework"
            value={framework}
            onChange={handleFramework}
          >
            {frameworks.map((framework) => {
              return <option key={framework}>{framework}</option>;
            })}
          </select>
        </div>
        <button type="submit" className="btn btn-block">
          submit
        </button>
      </form>
    </div>
  );
};
export default OtherInputs;
```

FormData API

```
import Starter from './tutorial/06-forms/starter/05-form-data.jsx';
```

JS Nuggets - FormData API

- a great solution when you have bunch of inputs
- inputs must have name attribute

The FormData interface provides a way to construct a set of key/value pairs representing form fields and their values, which can be sent using the fetch() or XMLHttpRequest.send() method. It uses the same format a form would use if the encoding type were set to "multipart/form-data".

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

const UncontrolledInputs = () => {
  const [value, setValue] = useState(0);

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

    const formData = new FormData(e.currentTarget);
    // const name = formData.get('name');
    // console.log(name);
    // console.log([...formData.entries()]);
    const newUser = Object.fromEntries(formData);
    // do something (post request, add to list, etc)
    console.log(newUser);
    // Gotcha - re-render won't clear out the values
    setValue(value + 1);
    // reset values
    e.currentTarget.reset();
  };
  return (
    <div>
      <form className="form" onSubmit={handleSubmit}>
        <h4>Form Data API</h4>
        {/* name */}
        <div className="form-row">
          <label htmlFor="name" className="form-label">
            name
          </label>
          <input type="text" className="form-input" id="name"
name="name" />
        </div>
        {/* email */}
        <div className="form-row">
          <label htmlFor="email" className="form-label">
            Email
          </label>
          <input type="email" className="form-input" id="email"
name="email" />
        </div>
        {/* password */}
        <div className="form-row">
          <label htmlFor="password" className="form-label">
            Password
          </label>
          <input
            type="password"
            className="form-input"
            id="password"
            name="password"
          />
        </div>

        <button type="submit" className="btn btn-block">
          submit
        </button>
      </form>
    </div>
  );
};
export default UncontrolledInputs;
```

- e.currentTarget

In React, e.currentTarget returns the DOM element that triggered the event.

- Object From Entries

The Object.fromEntries() static method transforms a list of key-value pairs into an object.

```
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42],
]);

const obj = Object.fromEntries(entries);

console.log(obj);
// Expected output: Object { foo: "bar", baz: 42 }
```

- reset()

The reset() method is a built-in method in HTML that can be used to reset all form controls to their initial values. When this method is called on a form element, it will clear any user-entered data and reset the values of all form elements to their default values.

useRef

```
import Starter from './tutorial/07-useRef/starter/01-useRef-basics.jsx';
```

- DOES NOT TRIGGER RE-RENDER
- preserves the value between renders
- target DOM nodes/elements

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

const UseRefBasics = () => {
  const [value, setValue] = useState(0);
  const refContainer = useRef(null);

  console.log(refContainer);
  // {current:null}
  // set value ourselves or DOM node

  useEffect(() => {
    // console.log(refContainer.current);
    refContainer.current.focus();
  });

  const isMounted = useRef(false);

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(refContainer.current);
    const name = refContainer.current.value;
    console.log(name);
  };

  useEffect(() => {
    if (!isMounted.current) {
      isMounted.current = true;
      return;
    }
    console.log('re-render');
  }, [value]);

  return (
    <div>
      <form className="form" onSubmit={handleSubmit}>
        <div className="form-row">
          <label htmlFor="name" className="form-label">
            Name
          </label>
          <input
            type="text"
            id="name"
            ref={refContainer}
            className="form-input"
          />
        </div>
        <button type="submit" className="btn btn-block">
          submit
        </button>
      </form>
      <h1>value : {value}</h1>
      <button onClick={() => setValue(value + 1)} className="btn">
        increase
      </button>
    </div>
  );
};

export default UseRefBasics;
```

Custom Hooks

```
import Starter from './tutorial/08-custom-hooks/starter/01-toggle.jsx';
```

- same rules as regular hooks
- simplify component (less code)
- re-use functionality

useToggle.js

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

const useToggle = (defaultValue) => {
  const [show, setShow] = useState(defaultValue);
  const toggle = () => {
    setShow(!show);
  };
  return { show, toggle };
};

export default useToggle;
```

- Challenge
- in App.jsx import 02-fetch-data
- take a look at the component
- and try to setup custom fetch hook
- hint :
  hook should return isLoading,isError,user
  and take url as parameter

useFetchPerson.js

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

const useFetchPerson = (url) => {
  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);
  const [user, setUser] = useState(null);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const resp = await fetch(url);
        // console.log(resp);
        if (!resp.ok) {
          setIsError(true);
          setIsLoading(false);
          return;
        }

        const user = await resp.json();
        setUser(user);
      } catch (error) {
        setIsError(true);
        // console.log(error);
      }
      // hide loading
      setIsLoading(false);
    };
    fetchUser();
  }, []);

  return { isLoading, isError, user };
};

export default useFetchPerson;
```

Generic Fetch

useFetch.js

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

const useFetch = (url) => {
  const [isLoading, setIsLoading] = useState(true);
  const [isError, setIsError] = useState(false);
  // change state value
  const [data, setData] = useState(null);

  useEffect(() => {
    // change name
    const fetchData = async () => {
      try {
        const resp = await fetch(url);

        if (!resp.ok) {
          setIsError(true);
          setIsLoading(false);
          return;
        }
        // change to response
        const response = await resp.json();
        setData(response);
      } catch (error) {
        setIsError(true);
        // console.log(error);
      }
      // hide loading
      setIsLoading(false);
    };
    // invoke fetch data
    fetchData();
  }, []);

  return { isLoading, isError, data };
};
```

```
export default useFetch;
```

Context API

```
import Starter from './tutorial/09-context-api/starter';
```

Challenge

- create three components and nest them in such way :
- Navbar.jsx
    - NavLinks.jsx (nested in Navbar)
        - UserContainer.jsx (nested in NavLinks)
- import Navbar.jsx in App.jsx (remove container - CSS)
- in Navbar.jsx setup
    - user state value
        - default value {name:'something'}
    - logout function
        - set user back to null
- pass both of them down to UserContainer.jsx
- display user and button
- on button click set user back to null
- extra challenge
- if user null, in UserContainer display

  please login

Navbar.jsx

```
import { useState } from 'react';
import NavLinks from './NavLinks';

const Navbar = () => {
  const [user, setUser] = useState({ name: 'bob' });
  const logout = () => {
    setUser(null);
  };
  return (
    <nav className="navbar">
```

```
        <h5>CONTEXT API</h5>
        <NavLinks user={user} logout={logout} />
      </nav>
    );
  };
  export default Navbar;
```

NavLinks.jsx

```
import UserContainer from './UserContainer';

const NavLinks = ({ user, logout }) => {
  return (
    <div className="nav-container">
      <ul className="nav-links">
        <li>
          <a href="#">home</a>
        </li>
        <li>
          <a href="#">about</a>
        </li>
      </ul>
      <UserContainer user={user} logout={logout} />
    </div>
  );
};
export default NavLinks;
```

UserContainer.jsx

```
const UserContainer = ({ user, logout }) => {
  return (
    <div className="user-container">
      {user ? (
        <>
          <p>Hello There, {user.name.toUpperCase()}</p>
          <button type="button" className="btn" onClick={logout}>
            logout
          </button>
        </>
      ) : (
        <p>Please Login</p>
      )}
    </div>
  );
};
export default UserContainer;
```

Setup Global Context

final code in the repo under w-assets

- create new VITE project

```
npm create vite@latest global-context -- --template react
```

- install and start the project

```
npm install && npm run dev
```

- in src create context.jsx
- setup a global context - GlobalContext
- setup a component (AppContext) with one state value
- return GlobalContext.Provider from AppContext
- wrap then entire application (main.jsx) - children prop "gotcha"
- setup a custom hook
- access in App.jsx
- log result

useReducer

```
import Starter from './tutorial/10-useReducer/starter/01-
useReducer.jsx';
```

- it's the complete file from 03-useState-array

Challenge

- let's add reset functionality
- create function that set's people back to data array
- create another button, similar to clear just for reset
- use conditional rendering to toggle between the buttons,
  depending on people value

```
const resetList = () => {
  setPeople(data);
};

// JSX
{
  people.length < 1 ? (
    <button className="btn" style={{ marginTop: '2rem' }} onClick=
{resetList}>
      reset
    </button>
  ) : (
    <button className="btn" style={{ marginTop: '2rem' }} onClick=
{clearList}>
      clear
    </button>
  );
}
```

```
import React from 'react';
import { data } from '../../../data';
const ReducerBasics = () => {
  const [people, setPeople] = React.useState(data);

  const removeItem = (id) => {
    let newPeople = people.filter((person) => person.id !== id);
    setPeople(newPeople);
  };
  const resetList = () => {
    setPeople(data);
  };
  return (
    <div>
      {people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
            <button onClick={() => removeItem(id)}>remove</button>
          </div>
        );
      })}
      {people.length < 1 ? (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={resetList}
        >
          reset
        </button>
      ) : (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={clearList}
        >
          clear
        </button>
      )}
    </div>
  );
};

export default ReducerBasics;
```

Remove useState

```
import { useState, useReducer } from 'react';
import { data } from '../../../data';

// default/initial state
const defaultState = {
  people: data,
};
// reducer function
// whatever state is returned from the function is the new state

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

// dispatch({type:'SOME_ACTION'}) an action
// handle it in reducer, return new state

const ReducerBasics = () => {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {
    // let newPeople = people.filter((person) => person.id !== id);
    // setPeople(newPeople);
  };

  const clearList = () => {
    // setPeople([]);
  };
  const resetList = () => {
    // setPeople(data);
  };

  return (
    <div>
      {/* switch to state */}
      {state.people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
```

```
              <button onClick={() => removeItem(id)}>remove</button>
            </div>
          );
        })}
        {/* switch to state */}
        {state.people.length < 1 ? (
          <button
            className="btn"
            style={{ marginTop: '2rem' }}
            onClick={resetList}
          >
            reset
          </button>
        ) : (
          <button
            className="btn"
            style={{ marginTop: '2rem' }}
            onClick={clearList}
          >
            clear
          </button>
        )}
      </div>
    );
  };

export default ReducerBasics;
```

First Dispatch

```
import { useState, useReducer } from 'react';
import { data } from '../../../data';

const defaultState = {
  people: data,
  isLoading: false,
};

const reducer = (state, action) => {
  if (action.type === 'CLEAR_LIST') {
    return { ...state, people: [] };
  }
};

const ReducerBasics = () => {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {
    // let newPeople = people.filter((person) => person.id !== id);
    // setPeople(newPeople);
  };

  const clearList = () => {
    dispatch({ type: 'CLEAR_LIST' });
    // setPeople([]);
  };
  const resetList = () => {
    // setPeople(data);
  };
  console.log(state);
  return (
    <div>
      {state.people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
            <button onClick={() => removeItem(id)}>remove</button>
          </div>
        );
      })}
      {state.people.length < 1 ? (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={resetList}
        >
          reset
        </button>
      ) : (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={clearList}
        >
          clear
        </button>
      )}
    </div>
  );
};

export default ReducerBasics;
```

Actions and Default State

```
import { useReducer } from 'react';
import { data } from '../../../data';

const CLEAR_LIST = 'CLEAR_LIST';
const RESET_LIST = 'RESET_LIST';
const REMOVE_ITEM = 'REMOVE_ITEM';

const defaultState = {
  people: data,
};

const reducer = (state, action) => {
  console.log(action);
  if (action.type === CLEAR_LIST) {
    return { ...state, people: [] };
  }

  throw new Error(`No Matching "${action.type}" - action type`);
};

const ReducerBasics = () => {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {};

  const clearList = () => {
    dispatch({ type: CLEAR_LIST });
  };

  const resetList = () => {};
  return (
    <div>
      {/* switch to state */}
      {state.people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
            <button onClick={() => removeItem(id)}>remove</button>
          </div>
        );
      })}
      {/* switch to state */}

      {state.people.length < 1 ? (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={resetList}
        >
          reset
        </button>
      ) : (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={clearList}
        >
          clear
        </button>
      )}
    </div>
  );
};

export default ReducerBasics;
```

Reset List Challenge

- setup a dispatch and handle action in the reducer

```
import { useReducer } from 'react';
import { data } from '../../../data';

const CLEAR_LIST = 'CLEAR_LIST';
const RESET_LIST = 'RESET_LIST';
const REMOVE_ITEM = 'REMOVE_ITEM';

const defaultState = {
  people: data,
};

const reducer = (state, action) => {
  console.log(action);
  if (action.type === CLEAR_LIST) {
    return { ...state, people: [] };
  }
  if (action.type === RESET_LIST) {
    return { ...state, people: data };
  }
  throw new Error(`No Matching "${action.type}" - action type`);
};

const ReducerBasics = () => {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {};

  const clearList = () => {
    dispatch({ type: CLEAR_LIST });
  };
  const resetList = () => {
    dispatch({ type: RESET_LIST });
  };

  return (
    <div>
```

```
        {/* switch to state */}
        {state.people.map((person) => {
          const { id, name } = person;
          return (
            <div key={id} className="item">
              <h4>{name}</h4>
              <button onClick={() => removeItem(id)}>remove</button>
            </div>
          );
        })}
        {/* switch to state */}

        {state.people.length < 1 ? (
          <button
            className="btn"
            style={{ marginTop: '2rem' }}
            onClick={resetList}
          >
            reset
          </button>
        ) : (
          <button
            className="btn"
            style={{ marginTop: '2rem' }}
            onClick={clearList}
          >
            clear
          </button>
        )}
      </div>
    );
  };

  export default ReducerBasics;
```

Remove Person Challenge

- remove single person
- hint extra property in the object

```
import { useReducer } from 'react';
import { data } from '../../../data';

const CLEAR_LIST = 'CLEAR_LIST';
const RESET_LIST = 'RESET_LIST';
const REMOVE_ITEM = 'REMOVE_ITEM';

const defaultState = {
  people: data,
};
```

```
const reducer = (state, action) => {
  console.log(action);
  if (action.type === CLEAR_LIST) {
    return { ...state, people: [] };
  }
  if (action.type === RESET_LIST) {
    return { ...state, people: data };
  }
  if (action.type === REMOVE_ITEM) {
    let newPeople = state.people.filter(
      (person) => person.id !== action.payload.id
    );

    return { ...state, people: newPeople };
  }

  return state;
};

const ReducerBasics = () => {
  const [state, dispatch] = useReducer(reducer, defaultState);

  const removeItem = (id) => {
    dispatch({ type: REMOVE_ITEM, payload: { id } });
  };

  const clearList = () => {
    dispatch({ type: CLEAR_LIST });
  };
  const resetList = () => {
    dispatch({ type: RESET_LIST });
  };

  return (
    <div>
      {/* switch to state */}
      {state.people.map((person) => {
        const { id, name } = person;
        return (
          <div key={id} className="item">
            <h4>{name}</h4>
            <button onClick={() => removeItem(id)}>remove</button>
          </div>
        );
      })}
      {/* switch to state */}

      {state.people.length < 1 ? (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={resetList}
        >
          reset
        </button>
      ) : (
        <button
          className="btn"
          style={{ marginTop: '2rem' }}
          onClick={clearList}
        >
          clear
        </button>
      )}
    </div>
  );
};

export default ReducerBasics;
```

Import / Export

- create new file - actions.js
    - copy/paste all actions
    - export/import actions
- create new file - reducer.js
    - copy/paste reducer
    - import actions
    - import data
    - export/import reducer

Performance

Lower State / Push The State Down

```
import Starter from './tutorial/11-performance/starter/01-lower-state';
```

When Component Re-Renders :

- When the component's state or props change, React will re-render the component to reflect these changes.
- When the parent element re-renders, even if the component's state or props have not changed.
- lower state

```
import { useState } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <button
      className="btn"
      onClick={() => setCount(count + 1)}
      style={{ marginBottom: '1rem' }}
    >
      count {count}
    </button>
  );
};
export default Counter;
```

Lower State Challenge

```
import Starter from './tutorial/11-performance/starter/02-lower-state-
challenge';
```

- fix the re-rendering
- hint addPerson fix

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

const Form = ({ addPerson }) => {
  const [name, setName] = useState('');
  const handleSubmit = (e) => {
    e.preventDefault();
    if (!name) {
      alert('Please Provide Name Value');
      return;
    }
    addPerson(name);
    setName('');
  };
  return (
    <form className="form" onSubmit={handleSubmit}>
      <div className="form-row">
        <label htmlFor="name" className="form-label">
          name
        </label>
        <input
          type="text"
          name="name"
          id="name"
          className="form-input"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <button className="btn btn-block" type="submit">
        submit
      </button>
    </form>
  );
};
export default Form;
```

React.memo()

```
import Starter from './tutorial/11-performance/starter/03-hooks';
```

React.memo is a higher-order component (HOC) in React that allows you to memoize a component. This means that if the input props to the component have not changed, the memoized component will return the same result from the previous render, instead of re-rendering. This can help improve performance by avoiding unnecessary render cycles.

The React.memo function takes a functional component as its argument and returns a new component that has the same behavior, but with the added optimization of checking if the props have changed. If the props have not changed, the memoized component will return the cached result from the previous render.

Here's an example of using React.memo

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

React.memo(Component) - returns memoized component

Function "Gotcha"

- setup remove person function

```
const removePerson = (id) => {
  const newPeople = people.filter((person) => person.id !== id);
  setPeople(newPeople);
};
```

- pass it down to List and Person

UseCallback

The useCallback hook is a hook in React that allows you to memoize a function. It takes two arguments: the first is the function you want to memoize, and the second is an array of dependencies. The hook will return a memoized version of the function that only changes if one of the values in the dependency array changes.

By memoizing the function, you can avoid unnecessary re-renders and improve the performance of your React application. The function will only be re-created if one of its dependencies changes, otherwise the same instance of the function will be returned. This can be useful in situations where you have an expensive function that you only want to recompute when its dependencies change.

Here is an example of how you might use useCallback:

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

function MyComponent() {
  const [data, setData] = useState([]);
  const handleClick = useCallback(() => {
    console.log(data);
  }, [data]);

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

In this example, the handleClick function is memoized using useCallback and the data prop is passed as a dependency. This means that the handleClick function will only be re-created if the data prop changes.

useCallback - Common Use Case

```
import Final from './tutorial/02-useEffect/final/04-fetch-data';
```

```
import { useState, useEffect, useCallback } from 'react';
const url = 'https://api.github.com/users';

const FetchData = () => {
  const [users, setUsers] = useState([]);
  const fetchData = useCallback(async () => {
    try {
      const response = await fetch(url);
      const users = await response.json();
      setUsers(users);
    } catch (error) {
      console.log(error);
    }
  }, []);

  useEffect(() => {
    fetchData();
  }, [fetchData]);
  // rest of the logic
};
```

useMemo

The useMemo hook is a hook in React that allows you to memoize a value. It takes two arguments: the first is a function that returns the value you want to memoize, and the second is an array of dependencies. The hook will return the memoized value that will only change if one of the values in the dependency array changes.

By memoizing a value, you can avoid unnecessary calculations and improve the performance of your React application. The value will only be recalculated if one of its dependencies changes, otherwise the same instance of the value will be returned. This can be useful in situations where you have an expensive calculation that you only want to recompute when its dependencies change.

Here is an example of how you might use useMemo:

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

function MyComponent({ data }) {
  const processedData = useMemo(() => {
    return data.map((item) => item.toUpperCase());
  }, [data]);

  return (
    <div>
      {processedData.map((item) => (
        <div key={item}>{item}</div>
      ))}
    </div>
  );
}
```

In this example, the processedData value is memoized using useMemo and the data prop is passed as a dependency. This means that the processedData value will only be recalculated if the data prop changes.

- create slowFunction file
- setup a function

- import in index.js and set it equal to a value

```
const slowFunction = () => {
  let value = 0;
  for (let i = 0; i <= 1000000000; i++) {
    value += i;
  }
  return value;
};

export default slowFunction;
```

useTransition

JS Nuggets - Array.from

```
import Starter from './tutorial/11-performance/starter/04-react-18';
```

- useTransition is a React Hook that lets you update the state without blocking the UI.

```
import { useState, useTransition } from 'react';
const LatestReact = () => {
  const [text, setText] = useState('');
  const [items, setItems] = useState([]);
  const [isPending, startTransition] = useTransition();

  const handleChange = (e) => {
    setText(e.target.value);

    startTransition(() => {
      const newItems = Array.from({ length: 5000 }, (_, index) => {
        return (
          <div key={index}>
            <img src="/vite.svg" alt="" />
          </div>
        );
      });
      setItems(newItems);
    });
  };
  return (
    <section>
      <form className="form">
        <input
          type="text"
          className="form-input"
          value={text}
          onChange={handleChange}
        />
      </form>
      <h4>Items Below</h4>
      {isPending ? (
        'Loading...'
      ) : (
        <div
          style={{
            display: 'grid',
            gridTemplateColumns: '1fr 1fr 1fr',
            marginTop: '2rem',
          }}
        >
          {items}
        </div>
      )}
    </section>
  );
};
export default LatestReact;
```

Suspense API

The Suspense API is a feature in React that allows you to manage the loading state of your components. It provides a way to "suspend" rendering of a component until some data has been fetched, and display a fallback UI in the meantime. This makes it easier to handle asynchronous data loading and provide a smooth user experience in your React application.

Here is an example of how you might use the Suspense API:

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

const DataComponent = lazy(() => import('./DataComponent'));

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

```
import { useState, useTransition, lazy, Suspense } from 'react';
const SlowComponent = lazy(() => import('./SlowComponent'));
const LatestReact = () => {
  const [text, setText] = useState('');
```

```
    const [items, setItems] = useState([]);
    const [isPending, startTransition] = useTransition();
    const [show, setShow] = useState(false);
    const handleChange = (e) => {
      setText(e.target.value);

      startTransition(() => {
        const newItems = Array.from({ length: 5000 }, (_, index) => {
          return (
            <div key={index}>
              <img src="/vite.svg" alt="" />
            </div>
          );
        });
        setItems(newItems);
      });
    };
    return (
      <section>
        <form className="form">
          <input
            type="text"
            className="form-input"
            value={text}
            onChange={handleChange}
          />
        </form>
        <h4>Items Below</h4>
        {isPending ? (
          'Loading...'
        ) : (
          <div
            style={{
              display: 'grid',
              gridTemplateColumns: '1fr 1fr 1fr',
              marginTop: '2rem',
            }}
          >
            {items}
          </div>
        )}
        <button onClick={() => setShow(!show)} className="btn">
          toggle
        </button>
        {show && (
          <Suspense fallback={<h4>Loading...</h4>}>
            <SlowComponent />
          </Suspense>
        )}
      </section>
    );
  };
  export default LatestReact;
```

- typical setup (wrap entire return in Suspense)

```
return (
  <Suspense fallback={<h4>Loading...</h4>}>
    {/* rest of the logic */}
    <section>{show && <SlowComponent />}</section>
  </Suspense>
);
```