const **UseStateObject** = () => {

  const [person, **setPerson**] = **React**.**useState**({

    name: "chirudeep",

    age: 24,

    message: "random message",

  });

  console.**log**(person);

  function **clickHandler**() {

**setPerson**({name: "karthik" });

  }

*// setPerson({ name: "chirud", age: 24, message: "new message" });*

  return (

    <>

      <h1>{person.name}</h1>

      <button *onClick*={**clickHandler**} *className*="btn">

        click this for change

      </button>

    </>

  );

};

Here setPerson sets the person object to { name: "karthik" } .

To set all fields, we have to send entire object like this:

setPerson({name:”karthik”,age:24,message:”random message”}).

Here even if we have to change a single field, we have to destructure entire object.

But a short cut to do is using spread out operator :

**setPerson**({ ...person, name: "karthik" });

This statement is equivalent to :

setPerson({name:”karthik”,age:24,message:”random message”}).

setPerson({copy old values using spread operator, value you want to change})

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setValue**(value + 1);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

      <button

*onClick*={() => {

**setValue**(value - 1);

        }}

*class*="btn"

      >

        decrement

      </button>

      <button

*onClick*={() => {

**setValue**(0);

        }}

*class*="btn"

      >

        Reset

      </button>

    </>

  );

};

The following code takes 2sec before setting the new value.

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setTimeout**(() => {

**setValue**(value + 1);

    }, 2000);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

    </>

  );

};

Here setTimeout takes to arguments

1—inline function which needs to be executed after time duration t.

2—time duration t.

If setTimeout is invoked, the function passed as parameter is executed after t milliseconds.

But here, even if we click the increment button multiple times, it will increment only once.

This is because setValue is asynchronous.

i.e, the setValue function takes the old value before increment and increments it instead of incrementing the updated value.

setValue takes two types of arguments:

1—new value to be set

2—inline function which returns the new value to be set.

This function by default takes previous state value as parameter.

And this has to return the updated value.

const **UseStateCounter** = () => {

  var [value, **setValue**] = **React**.**useState**(0);

  function **clickHandler**() {

**setTimeout**(() => {

**setValue**((prevValue) => {

        return prevValue + 1;

      });

    }, 2000);

  }

  return (

    <>

      <h1>{value}</h1>

      <button *onClick*={**clickHandler**} *class*="btn">

        increment

      </button>

    </>

  );

};

Here, inline function in the setValue gets previous Value of state as input by default and returns the new value to be set.

useEffect—

By default, useEffect runs after every re-render

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button *className*="btn" *onClick*={() => **setValue**(value + 1)}>

        increment

      </button>

    </>

  );

};

Here every time we click increment button, useEffectBasics is rendered.

And every time useEffectBasics is rendered, inline function defined in useEffect is run.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    document.title = value;

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here every time button is clicked, it increments the value and component is rendered.

Here useEffect is called every time component is rendered and tab’s title is changed.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    if (value > 1) {

      document.title = value;

    }

  });

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here title of the page changes only after value becomes 2.

useEffect can have two arguments.

First is the function to be executed on each render, second is the array of dependencies.

If the list is empty, function runs only on initial render but not subsequent renders.

const **UseEffectBasics** = () => {

**useEffect**(() => {

    console.**log**("useeffect");

    document.title = value;

  }, []);

  var [value, **setValue**] = **useState**(0);

  return (

    <>

      <h1>{value}</h1>

      <button

*className*="btn"

*onClick*={() => {

**setValue**(value + 1);

        }}

      >

        increment

      </button>

    </>

  );

};

Here, title of page is set to 0 when we load the pages. Then it is not changed.

If we add value variable to list of dependencies, then the page title changes every time value variable changes

There can be multiple useEffect functions defined .

const **UseEffectCleanup** = () => {

  var [size, **setSize**] = **useState**(window.innerWidth);

function **eventHandler**() {

**setSize**(window.innerWidth);

  }

**useEffect**(() => {

    window.**addEventListener**("resize", **eventHandler**

    });

  });

  return (

    <>

      <h2>useEffect cleanup</h2>

      <h2>{size}</h2>

    </>

  );

};

Here we have added a event listener on window when the component renders first time.

This listens to changes in window size and sets the value of size variable(state) on change of the window size.

This triggers re render. Now useEffect is called again and a event listener on window is added again.

i.e, for every render, a new event listener is created on window.

This increases number of event listeners exponentially.

The syntax of useEffect is:

**useEffect**(() => {-----------function which runs on every render

    effect

    return () => {------------cleanup function-ran 1st

      cleanup

    }

  }, [input])

Here useEffect function has two arguments.

A function and a list of dependencies.

Function can have a return statement which can return a function which can perform cleanup operations .

This cleanup function runs before other operations of useEffect is ran.

const **UseEffectCleanup** = () => {

  var [size, **setSize**] = **useState**(window.innerWidth);

  function **eventHandler**() {

**setSize**(window.innerWidth);

  }

  const **cleanupfn** = () => {

    console.**log**("cleanup");

    window.**removeEventListener**("resize", **eventHandler**);

  };

**useEffect**(() => {

    console.**log**("useeffect");

    window.**addEventListener**("resize", **eventHandler**);

    return **cleanupfn**;

  });

Here the cleanup function is ran before other operations are done.

This removes the event listener previously defined thus preventing the exponential increase in number of event listeners.

Note: First render wont call cleanup function(i.e cleanup function won’t be called on page loading). This is because there Is nothing to clean on initial load.

Fetchdata with useEffect:

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

const **UseEffectFetchData** = () => {

  var [users, **setUsers**] = **useState**([]);

  function **DataGetter**() {

**fetch**(url)

      .**then**((response) => response.**json**())

      .**then**((data) => **setUsers**(data));

  }

**useEffect**(() => {

**DataGetter**();

  });

  return (

    <>

      <ul *className*="users">

        {users.**map**((user) => {

          return (

            <h1>{user.login}</h1>

          );

        })}

      </ul>

    </>

  );

};

Here useEffect calls DataGetter function which gets data with fetch operation.

This is converted to json and users is set to this data using setUsers.

But this has a problem.

We are calling DataGetter function in useEffect.

This DataGetter gets the data and changes the state of users variable.

This triggers rendering of component.

On rendering, we useEffect is invoked which again calls DataGetter.

DataGetter again changes the state which triggers render.

This causes a infinite loop of data fetching and rendering component.

useEffect🡪DataGetter()🡪fetchdata🡪setState of user variable 🡪Render component🡪useEffect.

To avoid this infinite loop, we place a empty list of dependencies, which makes the component to render only when the page is loaded instead of every change to state.

i.e,

const **UseEffectFetchData** = () => {

  var [users, **setUsers**] = **useState**([]);

  function **DataGetter**() {

**fetch**(url)

      .**then**((response) => response.**json**())

      .**then**((data) => **setUsers**(data));

  }

**useEffect**(() => {

**DataGetter**();

  },[]);

  return (

    <>

      <ul *className*="users">

        {users.**map**((user) => {

          return (

            <h1>{user.login}</h1>

          );

        })}

      </ul>

    </>

  );

};

But if we add users variable to the list of dependencies, the problem of infinite loop remains .

Because, now every change to users variable will trigger a render.

Read the following articles for understanding short circuiting:

<https://codeburst.io/javascript-short-circuit-conditionals-bbc13ac3e9eb>

<https://codeburst.io/javascript-short-circuit-evaluation-3709ffda6384>