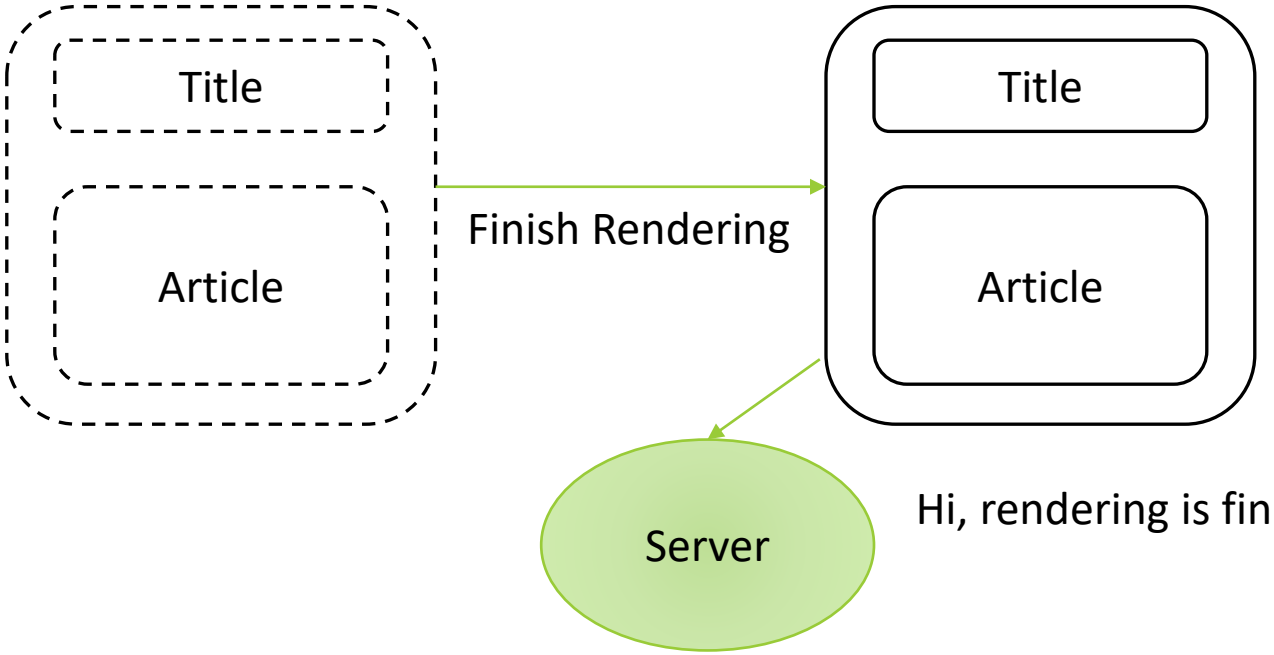
useEffect, custom hook, Virtual DOM Diffing

useEffect Hook

useEffect Hook

The useEffect is a React Hook function used to perform operations (side effects) in React components that are not triggered by events but by the rendering itself, such as sending AJAX requests, modifying the DOM, and so on.



In the diagram, there are no user events occurring. After the component is rendered, it needs to fetch data from the server. The entire process belongs to "operations triggered solely by rendering".

useEffect basic usage

Requirement: fetch data from server and display on the page once the component is rendered.

```
useEffect(() => { //Runs only on the first render }, []);
```

- Parameter 1 is a function, which can be called the effect function. Inside this function, you can place the operations to be executed.
- Parameter 2 is an array (optional), where you place dependencies. Different dependencies will affect the execution of the function passed as the first parameter. When it's an empty array, the effect function will only be executed once after the component is rendered.

Demo

```
const URL = 'https://tinaxing2012.github.io/jsons/data.json';

function App() {

  const [list, setList] = useState([]);

  useEffect(() => {
    async function getList(){
      const res = await fetch(URL);
      const data = await res.json();
      setList(data.data.channels);
    }
    getList();
  })
  return (
    <div>
      this is app
      <ul>
        {list.map((item: {id: number, name:string}) => <li key={item.id}>{item.name}</li>)}
      </ul>
    </div>
  );
}
```

useEffect dependencies

The execution of the useEffect effect function varies depending on the dependencies passed in. Different execution behaviors will occur based on the provided dependencies.

Dependencies	Execution of Effect function
No dependency passed	Runs on every render
An empty array	Runs only on the first render
Props or state values	Runs on first render + And any time any dependency value changes

useEffect Demo 1 – doesn't depend on anything

```
function App() {

  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log('Effect function is called');
  });

  return (
    <div>
      this is app
      <button onClick={() => setCount(count+1)}>+1</button>
    </div>
  );

}
```

useEffect Demo – depend on empty array []

```
function App() {

  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('Effect function is called');
  }, []);

  return (
    <div>
      this is app
      <button onClick={() => setCount(count+1)}>+1</button>
    </div>
  );

}
```

useEffect Demo 3
- depend on a variable

```
function App() {

  const [count, setCount] = useState(0);
  const [msg, setMsg] = useState('');

  useEffect(() => {
    console.log('Effect function is called');
  }, [count]);

  return (
    <div>
      this is app
      <button onClick={() => setCount(count+1)}>+1</button>
      <button onClick={() => setMsg(msg +'hi')}>Message</button>
    </div>
  );
}
```

Effect cleanup

Some effects require cleanup to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the `useEffect` Hook.

Note: The most common timing for executing effect cleanup is component unmounting.

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

Cleanup Demo

```
function Son(){
  useEffect(() => {
    const timerId = setInterval(() => {console.log('interval...')}, 1000);
    return () => {
      clearInterval(timerId);
    }
  }, []);
  return <div>This is Son!</div>
}

function App() {

  const [show, setShow] = useState(true);

  return (
    <div>
      this is app
      <button onClick={() => setShow(!show)}>Toggle</button>
      {show && <Son/>}
    </div>
  );

}
```

Custom Hooks

Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

1. Define a function starts with “use”
2. Encapsulate the usable logic inside function body
3. Return state and callback in the function

Demo

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

function App() {

  const {show, toggle} = useToggle();

  return (
    <div>
      {show && <div>this is app</div>}
      <button onClick={toggle}>Toggle</button>
    </div>
  );
}
```

Rules of React Hooks

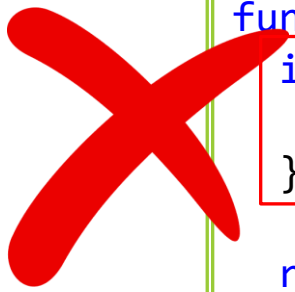
- It can only be called within a component or other custom Hook functions.
- It can only be called at the top level of a component and cannot be nested within if statements, for loops, or other functions.

```
const [show, setShow] = useState(true);

function App() {

  const toggle = () => setShow(!show);

  return (
    <div>
      {show && <div>this is app</div>}
      <button onClick={toggle}>Toggle</button>
    </div>
  );
}
```



```
function App() {
  if(Math.random() > 0.5){
    const [show, setShow] = useState(true);
  }

  return (
    <div>
      this is App
    </div>
  );
}
```

Virtual DOM Diffing

REACT

Demonstrating the Existence of React's Virtual DOM Diffing

In the Clock demo on the right-hand side.

When we input a value into the text input field, the `render()` method is triggered after one second, resulting in the update of only the time statement within the `<p>` tag, while the input fields remain unaffected.

If Virtual DOM Diffing were not in place, all elements, including the input fields, would be updated.

```
function App() {
  const [time, setTime] = useState<Date>(new Date());

  useEffect(() => {
    setInterval(() => {
      setTime(new Date());
    }, 1000);
  }, []);

  return (
    <div>
      <h1>DOM Diffing</h1>
      <input type='text'/>
      <p>Current time is: {time.toLocaleTimeString()} <input/></p>
    </div>
  )
}
```

Virtual DOM Diffing

In React, the Virtual DOM (VDOM) diffing is a core concept that contributes to the framework's efficiency and performance. The VDOM is a lightweight copy of the actual DOM, and React uses it to compare the current representation of a component with the next one to determine what changes need to be made.

What's the usage of "key" attribute in virtual DOM?

In React, the "`key`" attribute is used to help React identify and track individual elements in a list when rendering or updating components efficiently. It is not part of the Virtual DOM (VDOM) itself but plays a crucial role in how React reconciles changes in the real DOM using the VDOM.

How Diffing Algorithm Works?

When data in the state changes, React generates a new virtual DOM based on the updated data. Following this, React compares the new virtual DOM with the old virtual DOM using a diffing process.

Rules:

1. The same key as the new virtual DOM was found in the old virtual DOM
    1) If the content in the virtual DOM hasn't changed, the previous real DOM is used directly.
    2) If the content in the virtual DOM has changed, a new real DOM is generated and then replaces the previous real DOM on the page.
2. The same key as the new virtual DOM was NOT found in the old virtual DOM
    1) Create a new real DOM based on the data, and then render it on the page.

Any problem of using index as key?

In the Person demo on the right-hand side.

We add a new Person at the end of the array, then change the state.

What will occur if we instead insert the person at the beginning of the array?

```
function App() {
  const [person, setPerson] = useState([{ id: 1, name: 'John', age: 18 },
{ id: 2, name: 'Edward', age: 19 }]);

  const addPerson = () => {
    setPerson([
      ...person,
      {id: person.length, name: 'JJ', age: 20}
    ])
  }

  return (
    <div>
      <ul>
        {person.map((p, index) => <li key={index}>{p.name} {p.age}</li>)}
      </ul>
      <button onClick={addPerson}>add a new person</button>
    </div>
  )
}
```

The Issues That May Arise When Using Index as a Key

1. If operations on the data, such as reverse addition, reverse deletion, or any actions disrupting the order, are performed, unnecessary updates to the real DOM may occur. While the user interface remains unaffected, this can lead to low efficiency.
   - If the structure of JSX also includes DOM elements like input fields, it may lead to incorrect DOM updates, causing issues in the user interface.
2. If there's no operation like reverse addition to the data, there's no problem to use index as key.

Index as key issue - Demo

Add `newP` in the front of the array

An input field in the structure of JSX. VDOM diffing only compares the type of input filed, the value doesn't compare so real DOM is reused which cause doesn't match.

```
function App() {
  const [person, setPerson] = useState([{ id: 1, name: 'John', age: 18 }, {
id: 2, name: 'Edward', age: 19 }]);

  const addPerson = () => {
    setPerson([
      {id: person.length+1, name: 'JJ', age: 20},
      ...person
    ])
  }
  return (
    <div>
      <ul>
        {person.map((p, index) => <li key={index}>{p.name} {p.age}
<input/></li>)}
      </ul>
      <button onClick={addPerson}>add a new person</button>
    </div>
  )
}
```

Guidelines for Selecting Keys

1. It's best to use a unique identifier for each piece of data as the key, such as primary key, phone number, student ID, SSN, etc.
2. If you are certain that you're only displaying data, using the index is also acceptable.

```
function App() {
  const [person, setPerson] = useState([{ id: 1, name: 'John', age: 18 }, { id: 2,
name: 'Edward', age: 19 }]);

  const addPerson = () => {
    setPerson([
      {id: person.length + 1, name: 'JJ', age: 20},
      ...person
    ])
  }
  return (
    <div>
      <ul>
        {person.map((p, index) => <li key={p.id}>{p.name} {p.age} <input/></li>)}
      </ul>
      <button onClick={addPerson}>add a new person</button>
    </div>
  )
}
```

Main Point

React Hooks, introduced in React 16.8, are a set of functions that enable developers to add state and side-effect handling to functional components. This innovation allows for the management of component logic without the need for class components. By using hooks like useState and useEffect, developers can achieve cleaner, more readable code and better component reusability. Hooks have become a cornerstone of modern React development, simplifying state management and making it more approachable for both new and experienced developers.

The existence of a unified field of all the laws of nature, from which everything in the universe originates. It is described as a field of pure consciousness and the source of all knowledge and creativity.