요약

jsx

html과 유사하지만 바벨을 통해 js로 변환됨

jsx 규칙

1. 반드시 태그는 닫아야함

2. 2개 이상의 태그는 반드시 하나의 태그에 감싸져야함

++ fragment라는 <> </> 빈 태그를 감싸는 것을 통해 조건 준수가능

3. 해당 값의 내용을 보고싶다면 {}로 감싸면 됨

4. 기존의 - 을 쓰던것을 케멀백 스타일로 바꿔 써야함

5. 주석은 {/\* \*/}나 // 을 통해 사용

props

컴포넌트에 대해 파라미터 형태로 전달함

defaultprops

전달된 props가 없을때 기본으로 사용할 props

children

태그와 태그 사이에 들어가는 내용을 지칭

컴포넌트 내부에 다른 컴포넌트를 넣을 때 필요함

조건부 렌더링

isSpacial 이라는 별도의 props를 활용

react

import React from 'react';

import Hello from './hello';

function App() {

  return (

      <Hello name="react" isSpecial={true}/>

  );

}

export default App;

컴포넌트

import React from 'react';

function Hello(props) {

    return <div>

        {props.isSpecial ? <b>\*</b> : null }

        // {props.isSpecial && <b>\*</b>}

        // 보여주기만 하는 경우는 && 를 활용하는게 나음

        hi, {props.name}

    </div>;

}

export default Hello;

function Counter() {

    const onIncrease = () => {

        console.log('+1');

    }

    const onDecrease = () => {

        console.log('-1');

    }

    return (

        <div>

            <h1> 0 </h1>

            <button onClick={onIncrease}> +1 </button>

            <button onClick={onDecrease}> -1 </button>

        </div>

    )

}

클릭 시 변화하는 이벤트 함수

useState

동적 상태 반영

    const [number, setNumber] = useState(0);

    const onIncrease = () => {

        setNumber(number + 1);

    }

    const onDecrease = () => {

        setNumber(number - 1);

    }

Number 에 대해 setnumber 을 통해 값을 변경, useState는 최초값

Input 관리

import React, {useState} from 'react';

function InputSample () {

    const [text, setText] = useState('');

    const onChange = (e) => {

        setText(e.target.value);

    }

    const onReset = (e) => {

        setText('');

    }

    return (

        <div>

            <input onChange={onChange} value={text} />

            <button onClick={onReset}>초기화</button>

            <div>

                <b>값: </b>

                {text}

            </div>

        </div>

    )

}

export default InputSample;

**복수의 input**

import React, {useState} from 'react';

function InputSample () {

    const [inputs, setInputs] = useState({

        name: '',

        nickname: '',

    });

    const { name, nickname } = inputs;

    const onChange = (e) => {

        const {name, value} = e.target;

        setInputs({

            ...inputs,

            [name]: value,

        });

    }

    const onReset = (e) => {

    }

    return (

        <div>

            <input name="name" placeholder="이름" onChange={onChange} value={name}/>

            <input name="nickname" placeholder="닉네임" onChange={onChange} value={nickname}/>

            <button onClick={onReset}>초기화</button>

            <div>

                <b>값: </b>

                {name} ({nickname})

            </div>

        </div>

    )

}

export default InputSample;

복수의 경우 객체 형태의 useState를 통해 처리

비구조할당을 통해 객체의 속성을 상수로 변환

값 변경의 경우 스프레드 문법을 통해 이전 상태 객체내용을 모두 가져오고, 해당 대상만 변경

값 변경에서도 변한 대상의 속성을 비구조할당을 통해 처리

**Dom에 직접 접근**

import React, {useState, useRef} from 'react';

function InputSample () {

    const [inputs, setInputs] = useState({

        name: '',

        nickname: '',

    });

    const nameInput = useRef();

    const { name, nickname } = inputs;

    const onChange = (e) => {

        const {name, value} = e.target;

        setInputs({

            ...inputs,

            [name]: value,

        });

    }

    const onReset = () => {

        setInputs({

            name: '',

            nickname: '',

        });

        nameInput.current.focus();

    }

    return (

        <div>

            <input

                name="name"

                placeholder="이름"

                onChange={onChange}

                value={name}

                ref={nameInput}

            />

            <input name="nickname" placeholder="닉네임" onChange={onChange} value={nickname}/>

            <button onClick={onReset}>초기화</button>

            <div>

                <b>값: </b>

                {name} ({nickname})

            </div>

        </div>

    )

}

export default InputSample;

useRef라는 객체를 생성해 대상이 되는 이경우 input에 해당 객체를 ref로 설정하고,

이를 통해 초기화 후 focus 되는 형태의 구조가 추가됨

이를 통해 Dom 객체에 대해 직접 접근을 진행

**배열 랜더링**

import React from 'react';

function User({user}) {

    return (

        <div>

            <b>{user.username}</b> <span>{user.id}</span>

        </div>

    )

}

function UserList() {

    const users = [

        {

            id: 1,

            username: 'a',

        },

        {

            id: 2,

            username: 'b',

        },

        {

            id: 3,

            username: 'c',

        },

    ]

    return (

        <div>

            {

                (

                    users.map(

                        user => (<User user={user} key={user.id} />)

                    )

                )

            }

        </div>

    )

}

export default UserList;

컴포넌트 파일 내부에 복수의 컴포넌트를 활용해 랜더링 처리

배열의 map 함수를 활용해, 숫자에 관계없이 여러 배열을 랜더링 가능

useRef 를 통한 컴포넌트 안의 변수 생성가능

컴포넌트 리렌더링 없이

유지되는 변수를 생성하기 위하여

외부 라이브러리, 등을 위해 사용

**배열 요소 추가**

Push나 splice 등을 활용하는게 아니라, 새로운 객체를 생성하고

1. 마찬가지로 스프레드 연산자를 통해
2. 혹은 concat을 통해 해당 사용자를 연결

해당 내용을 추가한 뒤 적용하는 방식

import React, { useRef, useState } from 'react';

import CreateUser from './CreateUser';

import UserList from './UserList';

function App() {

  const [inputs, setInputs] = useState({

    user: '',

    email: '',

  })

  const { username, email } = inputs;

  const onChange = e => {

    const { name, value } = e.target;

    setInputs({

      ...inputs,

      [name]: value

    })

  }

  const [users, setUsers] = useState([

    {

        id: 1,

        username: 'a',

        email: 'aaa',

    },

    {

        id: 2,

        username: 'b',

        email: 'vvv',

    },

    {

        id: 3,

        username: 'c',

        email: 'ccc',

    },

  ]);

  const nextId = useRef(4);

  const onCreate = () => {

    const user = {

      id : nextId.current,

      username,

      email,

    }

    setUsers(users.concat(user));

    setInputs({

      username: '',

      email: ''

    })

    console.log(nextId.current);

    nextId.current += 1;

  }

  return (

    <>

      <CreateUser

        username={username}

        email={email}

        onChange={onChange}

        onCreate={onCreate}/>

      <UserList users={users} />

    </>

  )

}

export default App;

**배열 항목 제거**

Filter 함수를 통한 처리

필터 함수는 해당 조건에 맞는 대상들만 새로 배열을 생성하는데

이를 마찬가지로 useState에 적용함으로써 처리가능

생성 == 스프레드 / 삭제 == 필터 / 변경 == 맵

**useEffect Hook**

마운트 되는 시점과 언마운트 되는 시점에 특정 작업을 진행 가능함

setInterval, setTimeout 의 설정 제거

라이브러리 인스턴스 제거 등…

    useEffect(()=> {

        console.log('컴포넌트가 화면에 나타남');

        return () => {

            console.log('컴포넌트가 사라짐');

        }

    }, []);

또 뒤의 변수 입력에 따라 변경, 변경 전 값을 보여줌

    useEffect(()=> {

        console.log('변경');

        return () => {

            console.log('변경 전');

        }

    }, [user]);

**useMemo Hook**

특정 값이 바뀐 경우만 실행하도록 하여 낭비를 막음

  const count = useMemo(() => counterActiveUsers(users), [users]);

[ ] 의 값이 바뀐 경우만 해당 내용이 실행

**useCallback Hook**

useMemo와 유사하게 특정 대상의 변화에 해당 함수를 호출하도록 함

**React.memo**

불필요한 리랜더링을 방지해서 낭비를 막음, 컴포넌트 성능 최적화

export default React.memo(CreateUser);

와 같이 export 함수를 묶고

const User = React.memo(function User({user, onRemove}) {

    const {username, email, id} = user;

    useEffect(()=> {

        console.log('변경');

        return () => {

            console.log('변경 전');

        }

    }, [user]);

    return (

        <div>

            <b>{username}</b> <span>{email}</span>

            <button onClick={() => onRemove(id)}>삭제</button>

        </div>

    )

});

처럼 묶는 형태로 활용하며, 변화한 대상에 대해서만 랜더링 진행

useReducer

특정 객체를 바탕으로 상태 기반 업데이트

Reducer == 상태를 업데이트 하는 함수, return을 통해 반환 받는 방식으로 동작

import React, { useReducer } from 'react';

function reducer(state, action) {

    switch (action.type) {

        case 'INCREMENT':

            return state +1;

        case 'DECREMENT':

            return state -1;

        default:

            throw new Error('Unhandled action');

    }

}

function Counter() {

    const [number, dispatch] = useReducer(reducer, 0);

    const onIncrease = () => {

        dispatch({

            type: 'INCREMENT'

        })

    };

    const onDecrease = () => {

        dispatch({

            type: 'DECREMENT'

        })

    };

    return (

        <div>

            <h1> {number} </h1>

            <button onClick={onIncrease}> +1 </button>

            <button onClick={onDecrease}> -1 </button>

        </div>

    )

}

동작에 대한 함수 reducer

호출 dispatch (action)

**Usestate vs UseReducer 차이**

전자 값이 단순, 적은 경우

후자 값이 다양한 형태, 많은 수

**Context API**

직접적으로 해당 컴포넌트에 변수 값을 넘겨주는 것이 가능하도록 함