2025.01.09.금

생성일	@2026년 1월 9일 오전 8:58
태그	React

목차

▼ 1) 이벤트 핸들링

- 함수 선언문

```
function handleChange() {
  console.log("change");
}

return (
  <input type="text" onChange={handleChange}/>
);
```

- 화살표 함수

```
const handleChange = () => {
  console.log("change");
};

return (
  <input type="text" onChange={handleChange} />
);
```

- 인라인 화살표 함수

```
<input
  type="text"
  onChange={(event) => {
    setText(event.target.value);
  }}
/>
```

- 잘못된 예시 - 함수 호출을 직접 전달하면 함수가 즉시 실행됨

```
const handleChange = (event) => {
  setText(event.target.value);
  console.log('Text updated: ', event.target.value);
};

return (
  <input
```

```
      type="text"
      onChange={handleChange()} // 이벤트를 듣지 않고 함수 바로 실행
    />
);
```

▼ 2) 리스트와 키

- list: 배열과 키를 사용하여 아이템을 수넛대로 모아놓은 것
- array: 변수나 객체를 하나의 변수로 묶어놓은 것
- key: 각 객체나 아이템을 구분할 수 있는 고유 값, 중복시 오류
- map

```
const items = ['apple', 'banana', 'cherry'];

return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
```

▼ 3) 리액트 폼 (React form)

- onChange: 입력 값이 변경될 때 발생하는 이벤트, 제어 컴포넌트에서 상태를 업데이트

  ```
  <input type="text" onChange={(e) => setValue(e.target.value)} />
  ```

- onSubmit: 폼이 제출될 때 발생하는 이벤트, 기본 제출 동작을 방지하려면 event.preventDefault() 사용

  ```
  <form onSubmit={(e) => handleSubmit(e)} />
  ```

- onBlur: 사용자가 입력 필드를 벗어날 때 발생하는 이벤트, 유효성 검사 수행할 때 사용

  ```
  <input type="text" onBlur={() => validateInput()} />
  ```

- select 태그 사용법

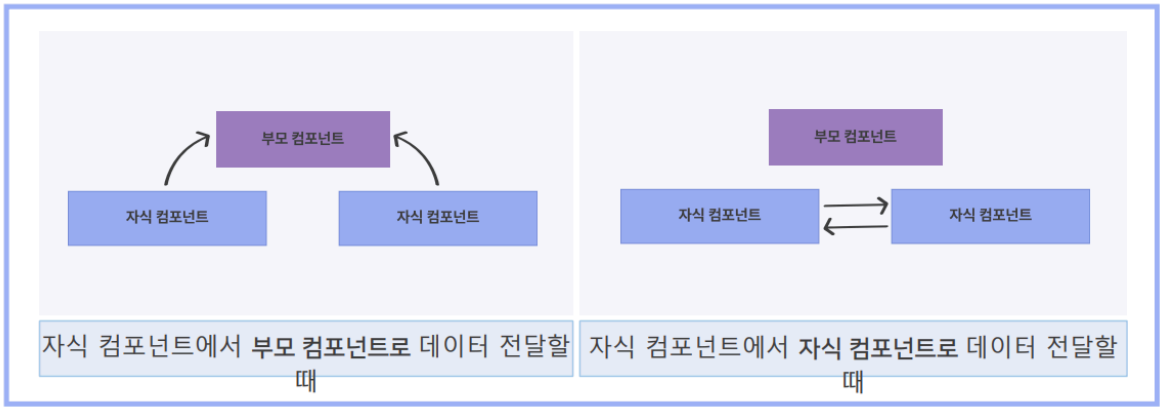
  ```
  const [selectedOption, setSelectedOption] = setState('1');

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

  return (
    <select value={selectedOption} onChange={handleChange}>
      <option value="1">Option 1</option>
      <option value="1">Option 1</option>
    </select>
  );
  ```

- 제어 컴포넌트
  - 리액트에서 입력 폼의 값을 상태(State)로 관리하는 컴포넌트
  - 입력한 값과 저장되는 값이 실시간으로 동기화됨
- 비제어 컴포넌트
  - useRef 훅을 사용하여 입력 필드 참조
  - 입력 필드의 값을 DOM에서 직접 읽음

- 값이 실시간으로 업데이트되지 않고 리렌더링 발생시키지 않음

▼ 4) State 상태 끌어올리기

- Shared State: State를 여러 컴포넌트에서 공통적으로 사용하는 경우



-

▼ 5) 합성과 상속

- 작은 컴포넌트들을 모아 더 큰 컴포넌트를 만드는 방식
- props.children을 사용한 합성

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}

function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

- 부모 컴포넌트가 하위 컴포넌트를 포함하는 방법

```
function Contacts() {
  return <div className="Contacts" />
}

function Chat() {
  return <div className="Chat" />
}

function SplitPane(props) {
```

```
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left = {
        <Contacts />
      }
      right = {
        <Chat />
      } />
  );
}
```

- 특수화 패턴: 범용적으로 쓸 수 있는 컴포넌트를 만들어 놓고 이를 특수화 시켜서 컴포넌트를 사용하는 합성 방식

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function welcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacercrat!" />
  );
}
```

▼ 6) 리액트 스타일링

- 바닐라 CSS
  - Styled-Components 라이브러리: JS 인라인 태그를 CSS 처럼 사용
- CSS Modules
  - .module.css 확장자를 통해 리액트 컴포넌트에 css 임포트
  - 컴포트된 컴포넌트 내에서만 유효함 (클래스에 고유한 해시 값을 붙여 구현됨)
  - :global 키워드를 사용하여 전역 스타일도 적용 가능하나 권장하지 않음
- Tailwind CSS
  - 유틸리티 클래스 기반의 접근 방식

    - 디자인 시스템 구축
- SCSS(SaSS)
    - CSS의 확장으로 변수, 중첩, 믹스인, 상속, 연산 등의 기능 지원

▼ 7) Redux

- Recoil
    - 리액트 컴포넌트 상태 관리 라이브러리
    - 비동기 상태 관리에 특화
- Zustand
    - 간단하고 가벼운 상태 관리 라이브러리
    - 적은 보일러 플레이트로 상태 관리
- MobX
    - 상태를 자동으로 반응형으로 만들어주는 도구
    - 상태 변화에 따른 동작을 따로 관리하지 않고 상태가 변할 때 자동으로 변경되는 것, observer를 사용해 상태를 감지
- Redux
    - js 애플리케이션의 상태를 중앙에서 관리하는 도구
    - 규모가 큰 프로젝트의 전역 상태 관리가 필요할 때 사용
    - 상태 관리의 복잡성이 증가하고 추적하는 것이 어려울 때 사용
    - 중앙에서 명시적으로 상태를 관리
- Redux 작동방식
    - Store
        - createStore를 통해 생성
        - 모든 상태를 하나의 스토어에서 관리

```
import { createStore } from 'redux';

const initialState = { count: 0 }; // 초기화

const store = createStore (counterReducer);

const counterReducer = {state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};
```

    - Action
        - 상태를 어떻게 변경할지 명령하는 객체
        - type 필드로 어떤 액션인지 식별
        - 추가 데이터는 payload 필드에 담아서 호출가능

```
const incrementAction = {
  type: 'INCREMENT',
  payload: 5
};
```

```
const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 + action.payload };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const updateUserAction = {
  type: 'UPDATE_USER',
  payload: {
    id: 1,
    name: 'kim',
    email: 'sang@examle.com'
  }
};
```

- dispatch
  - 이벤트 발생 시, 액션을 스토에 보내는 함수

```
const incrementAction = { type: 'INCREMENT' };

store.dispatch(incrementAction);
store.dispatch({ type: 'INCREMENT' });
```

- Redux와 Context API의 차이점

특징	리덕스(Redux)	리액트 컨텍스트(React Context)
주요 사용 사례	대규모 애플리케이션, 복잡한 상태 관리	테마, 인증 정보, 간단한 글로벌 상태
설정 복잡도	초기 설정이 복잡하고 보일러플레이트 많음	설정이 간단하고 직관적
상태 저장 방식	단일 스토어에서 중앙 집중식으로 관리	여러 컨텍스트로 분산 관리 가능
디버깅 도구	Redux DevTools로 상태 변화 추적 가능	디버깅 도구 부족
성능	상태 변화에 따라 필요한 컴포넌트만 렌더링	모든 컨텍스트 구독자가 리렌더링 됨
애플리케이션 크기	대규모 애플리케이션에 적합	소규모 애플리케이션에 적합