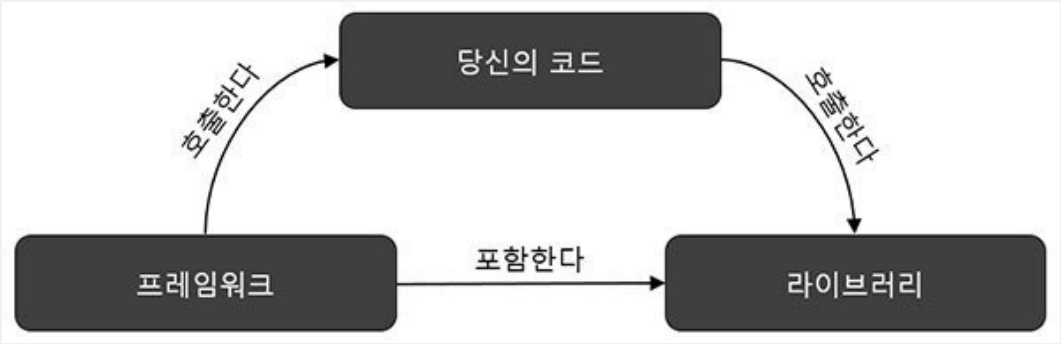
✔️ 2023.06.11 React & Redux

1. React

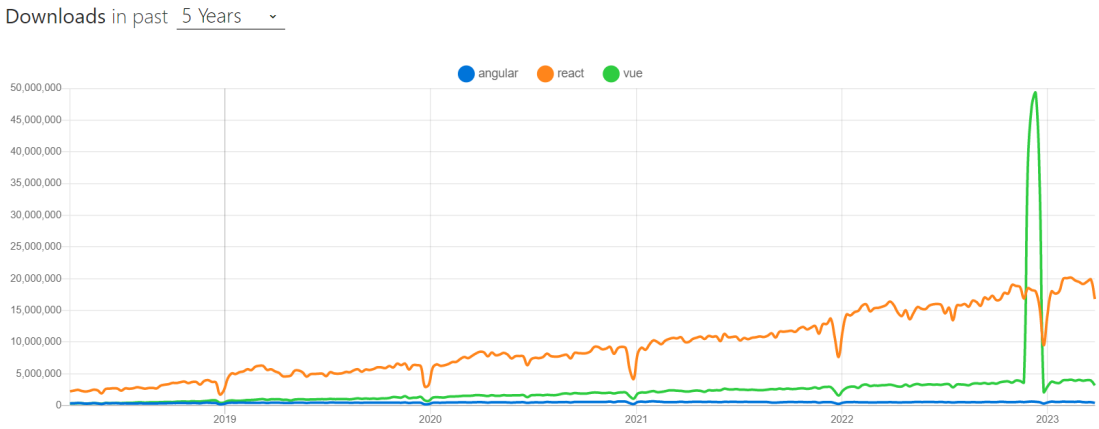
React는 UI를 만들기 위한 Javascript 라이브러리입니다. 라이브러리로 분류되는 이유는, 프로젝트를 수행하기 위한 모든 도구 세트를 갖추고 있지 않기 때문입니다. React 자체는 **UI와 관련된 일부 도구만 포함**하고 있습니다.

> ⚠️ 라이브러리와 프레임워크
> 1. 라이브러리: 활용 가능한 도구들의 집합을 의미합니다. 사용자가 **필요할 때마다 호출**하여 사용하는 것으로, 제어에 대한 **주도권은 사용자가** 가지고 있습니다.
>
> 2. 프레임워크: 소프트웨어의 특정 문제를 해결하기 위해서 상호 협력하는 **클래스와 인터페이스의 집합**입니다. 사용자는 미리 짜여진 구조에 맞게 코드를 작성해야 합니다. 따라서 제어에 대한 **주도권은 프레임워크가** 가지고 있습니다(**제어의 역전**).
>
> 
>

2. React를 사용하는 이유

1. 페이지 전체를 새로고침하지 않고 **변하는 부분만을 리렌더링하여 사용성을 향상**시켜 줍니다.
2. **컴포넌트 단위로 개발**이 가능하여 **가독성, 확장성, 재사용성을 향상**시켜 줍니다.
3. 라이브러리이므로 **다른 라이브러리 및 프레임워크와 함께 사용**할 수 있습니다.

4. 현실적인 이유로는 **가장 많이 사용**되는 SPA Tool이라는 점이 있습니다.



> ⚠️ SPA(**Single Page Application**)
> 서버로부터 완전한 새 페이지를 불러오지 않고, 현재 페이지를 동적으로 다시 작성함으로써 사용자와 소통하는 웹 애플리케이션이나 웹사이트를 의미합니다. 새로운 HTML 문서를 다시 호출하는 일이 없어 속도 및 사용자 경험이 향상됩니다.

3. Props & States

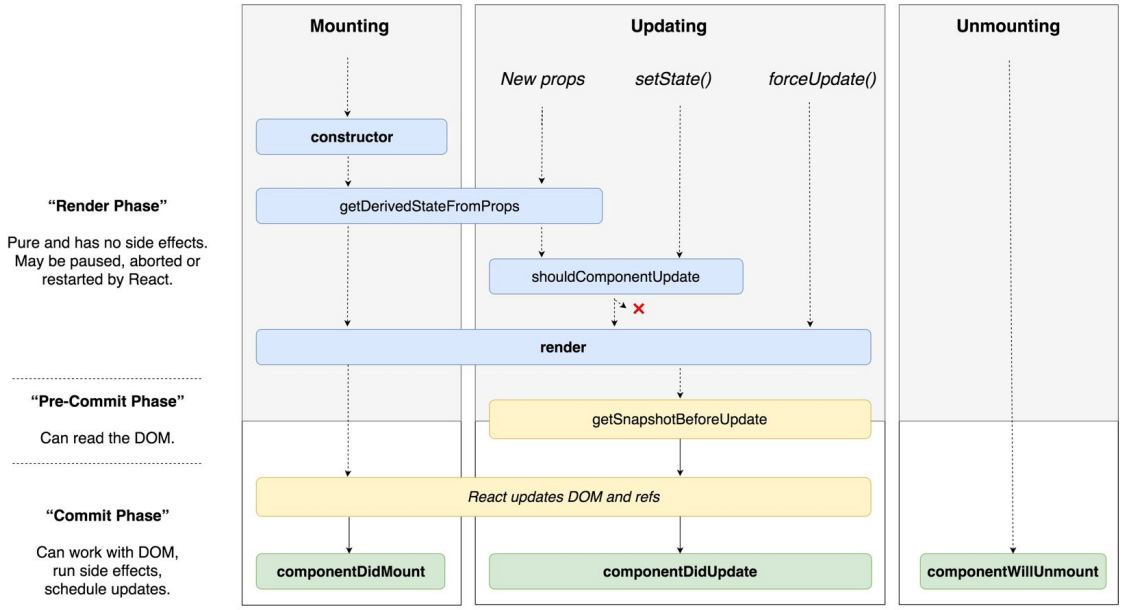
1. React는 사용자 정의 컴포넌트에서 **JSX attributes와 자식을 해당 컴포넌트에 단일 객체로 전달**하는데, 이를 props라고 합니다. 전달 받은 props는 **읽기 전용**입니다. (LWC: @api)
2. state는 props와 유사하지만 **비공개**이며, **해당 컴포넌트에 의해 완전히 제어**됩니다. 즉 컴포넌트 내에서 변경 가능한 값입니다.
3. setState는 **비동기적**으로 실행됩니다. 변경된 state를 한번에 처리하여 리렌더링 횟수를 줄이기 위해서입니다.
4. 부모 컴포넌트는 state를 자식 컴포넌트에 props로 전달할 수 있습니다.

4. Lifecycle

1. 클래스 컴포넌트에서의 생명 주기

   React의 생명 주기는 React가 실행되는 순서를 말합니다. 클래스 컴포넌트에서는 각각의 생명 주기마다 동작을 실행할 수 있도록 아래와 같은 생명 주기 함수를 제공했습니

다.



1. Mount Phase
    a. **컴포넌트가 처음 실행**되는 것을 Mount라고 합니다. 컴포넌트가 처음 실행되면 context, defaultProps, State를 저장한 뒤, **componentWillMount**를 호출합니다. 이 단계에서는 아직 DOM에 접근할 수 없고, props나 state를 바꾸는 것도 불가능합니다. (LWC: connectedCallback)
    b. 이후, render가 컴포넌트를 DOM에 부착하면 **Mount가 완료되고, componentDidMount**가 호출됩니다. (LWC: renderedCallback)

> ⚠️
> 1. context, defaultProps, state 저장
> 2. componentWillMount
> 3. render
> 4. componentDidMount

2. Update Phase (Props)
    a. Props가 업데이트 될 때의 과정입니다. **업데이트가 발생하면 componentWillRecieveProps**가 호출됩니다.
    b. sholudComponentUpdate가 호출됩니다. false를 반환하면 render를 취소할 수 있는데, 이를 통해 불필요한 render를 방지합니다.

c. componentWillUpdate가 호출됩니다. props를 수정하는 중이기 때문에 state를 수정할 수 없습니다.

d. render를 통해 **업데이트가 완료되면 componentDidUpdate**가 호출됩니다. DOM에 접근 가능합니다. (LWC: renderedCallback)

> ⚠️
> 1. componentWillRecieveProps
> 2. sholudComponentUpdate
> 3. componentWillUpdate
> 4. render
> 5. componentDidMount

3. Update Phase (State)

a. State가 업데이트 될 때의 과정입니다.

b. sholudComponentUpdate가 호출됩니다. false를 반환하면 render를 취소할 수 있는데, 이를 통해 불필요한 render를 방지합니다.

c. componentWillUpdate가 호출됩니다. props를 수정하는 중이기 때문에 state를 수정할 수 없습니다.

d. render를 통해 **업데이트가 완료되면 componentDidUpdate**가 호출됩니다. DOM에 접근 가능합니다. (LWC: renderedCallback)

> ⚠️
> 1. sholudComponentUpdate
> 2. componentWillUpdate
> 3. render
> 4. componentDidMount

4. Unmount Phase

a. 컴포넌트가 제거되는 것을 Unmount라고 합니다. unmount되기 전에 **componentWillUnmount**를 호출합니다. (LWC: disconnectedCallback)

2. 함수 컴포넌트에서의 생명 주기

함수 컴포넌트에서는 특정 데이터에 대해 생명 주기가 진행됩니다. useEffect Hook이 생명 주기 함수를 대체할 수 있습니다.

4. React Hook

- useState
  - 함수 컴포넌트에서 state를 사용할 수 있게 해 줍니다.
  - state로 설정한 변수는 함수의 실행이 끝나도 사라지지 않습니다.
  - useState는 인자로 state 변수의 초기 값을 가지며, state 변수와 이를 갱신할 수 있는 함수 쌍을 반환합니다(`[state, setState] = useState()`).
- useEffect
  - 컴포넌트가 렌더링 될 때마다(DOM이 업데이트 될 때마다) 부수 효과(Side Effect)를 실행해 줍니다.
  - useEffect는 첫 번째 인자로 side effect로서 수행될 함수를 가집니다.
    - side effect 함수 내에서 clean-up 함수를 반환하면, effect가 실행된 후 clean-up 함수를 실행합니다.
    - 클래스 컴포넌트에서 컴포넌트가 unmount될 때 한 번만 clean-up이 가능했던 것과는 달리, **모든 리렌더링 시에 clean-up이 발생**합니다.
    - 이는 컴포넌트가 화면에 표시되어 있는 동안 state가 변경되는 경우에 대응할 수 있게 해 줍니다. (접속 상태 변경 등)
  - useEffect는 두 번째 인자로 배열을 가질 수 있습니다.
    - 두 번째 인자에 특정 state 변수를 담은 배열을 전달하면, 해당 state가 변할 때마다 컴포넌트가 리렌더링되며 side effect가 실행됩니다.
    - 두 번째 인자에 빈 배열을 전달하면, 컴포넌트가 최초로 렌더링 된 후 side effect가 실행됩니다. clean-up 함수를 반환하는 경우, 컴포넌트가 unmount되는 경우에는 clean-up 함수가 실행됩니다.
    - 두 번째 인자로 아무것도 전달하지 않으면, 컴포넌트가 리렌더링 될 때마다 side effect가 실행됩니다.
- useContext
  - state 관리를 위한 React 자체 Hook으로, Props를 글로벌로 사용할 수 있습니다.
- useReducer
  - state 관리를 위한 React 자체 Hook으로, Redux의 Store와 같은 역할을 하는 컴포넌트를 만들 수 있습니다.
- useCallback

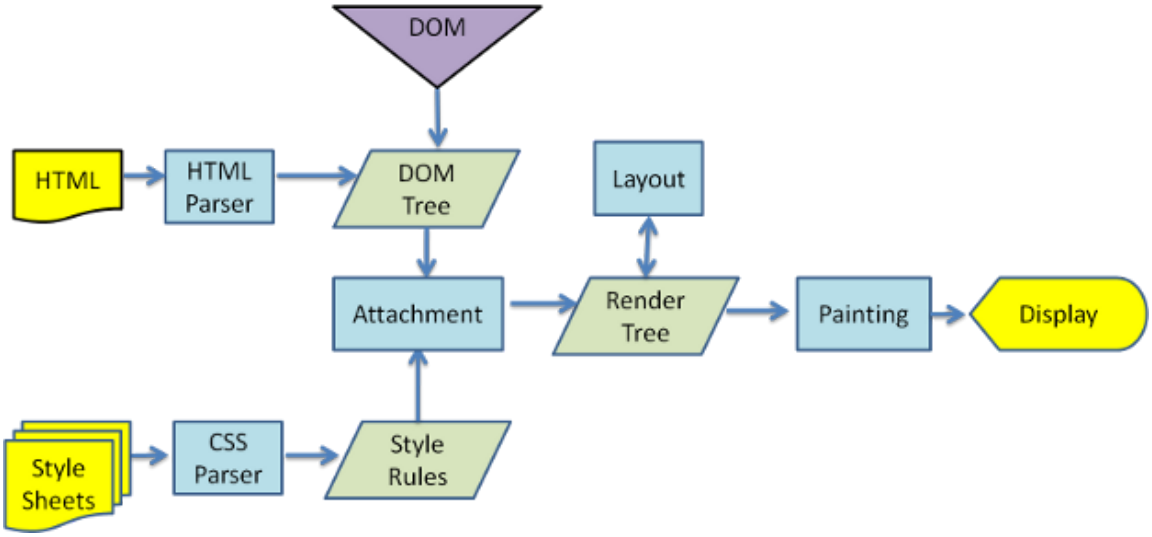
    - 특정 함수를 캐싱해 두었다가 재사용합니다. 컴포넌트가 리렌더링 될 때마다 함수가 새로 만들어지는 것을 방지합니다.
- useMemo
    - 특정 함수의 결과 값을 캐싱해 두었다가 재사용합니다. 컴포넌트가 리렌더링 될 때마다 함수가 새로 실행되는 것을 방지합니다.
- useRef
    - DOM에 접근해야 하는 경우 사용합니다.
    - react는 선언형 프로그래밍(what)이기 때문에, ref를 사용한 명령형 프로그래밍(how) 방식은 best practice가 아니므로 사용을 가능한 피하는 것이 좋습니다.
- useImperativeHandle
    - forwardRef를 사용하여 ref를 사용하는 부모 측에서 사용자 정의 함수를 사용할 수 있게 해 줍니다.
- useLayoutEffect
    - 동기적으로 부수 효과의 실행이 필요한 경우 사용합니다.
- useInsertionEffect
    - useLayoutEffect가 동작하기 전에 스타일을 먼저 조작하기 위해 사용합니다.
- useDebugValue
    - 어떤 컴포넌트에서 어떤 Hook을 사용했는지 확인할 수 있습니다.
- useTransition
    - 스로틀링이나 디바운싱을 대체하여 나온 Hook입니다. 리액트 엔진의 자체 스케줄러를 이용하여 무거운 연산보다 사용자의 상호작용을 우선적으로 처리합니다.

> 💡 **Throttling**은 마지막 함수가 호출된 후, 일정 시간이 지나기 전에 다시 호출되지 않도록 하는 것입니다.
> **Debouncing**은 연속적으로 호출되는 함수 중 마지막 함수 또는 처음 함수만 호출하도록 하는 것입니다.
> 두 방법 모두, 짧은 시간에 연속적으로 많은 이벤트가 발생하여 불필요한 요청이 발생하는 것을 방지하기 위한 프로그래밍 기법이지만, 근본적인 문제 해결책이 되지는 않습니다. 적절한 딜레이 시간을 설정하는 것도 모호하고, 부하가 큰 작업에는 적용해도 큰 의미가 없기 때문입니다.

- useDeferredValue
  - useTransition과 마찬가지로 낮은 우선순위를 지정하기 위한 것입니다. useTransition은 함수 실행의 우선순위를 지정하지만, useDeferredValue는 값의 업데이트 우선순위를 지정합니다.
- useId
  - 유니크한 Id를 만들어 줍니다. 단, 이는 반복문의 key로 사용되어서는 안됩니다.
- useSyncExternalStore
  - 외부 저장소를 구독할 수 있게 하는 Hook입니다.

5. Virtual DOM

1. 브라우저는 아래의 순서에 따라 동작합니다. HTML과 CSS로부터 각각 DOM Tree와 CSSOM Tree를 생성하여 결합(Attachment)하고, 이를 통해 Render Tree를 생성합니다.



2. DOM을 조작하면, 브라우저는 **매번** 스타일을 계산하고 실제 화면에 렌더링하는 작업을 수행합니다.
3. 그런데, Render Tree를 생성하는 과정에서, **DOM의 모든 요소의 스타일이 계산**됩니다.
4. 즉, DOM이 조작될 때마다 **변화하지 않는 요소를 포함한 모든 요소의 스타일이 다시 계산**되어 불필요한 비용이 발생하는 것입니다.
5. Virtual DOM의 역할

a. Virtual DOM은 말하자면 **오프라인 DOM**과 같습니다. 먼저 오프라인 DOM에서 연산을 수행하고, **최종적인 변화를 실제 DOM에 한 번만 전달**합니다.

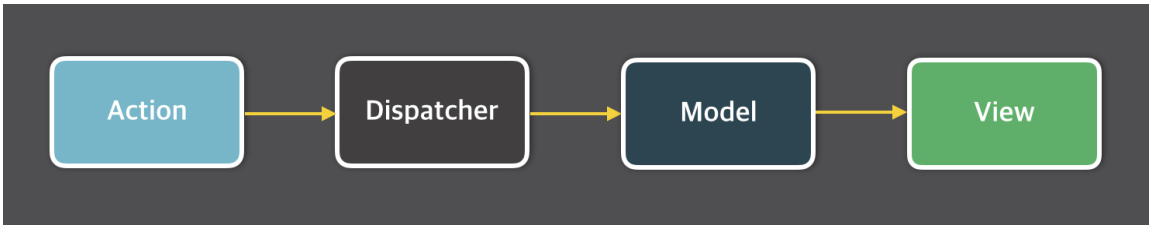
b. 위 과정은 사실 직접 수행할 수도 있습니다. 변화가 발생할 때, DOM fragment에 해당 변화를 전달하고, 이후 기존 DOM에 이를 보내면 된다고 합니다. 그러나 이를 직접 시도하면 기존 요소들 중 **어떤 것들이 변화되었고 어떤 것들이 변화되지 않았는지를 항상 파악**하고 있어야 합니다. Virtual DOM은 이러한 작업을 자동으로 해주는 역할 또한 수행하고 있습니다.

6. Redux

1. Redux를 사용하는 이유

   state는 기본적으로 컴포넌트에 종속되어 있습니다. 그러나 로그인 여부 등과 같이, 프로젝트 전체에서 사용해야 하는 state도 존재하고, props를 통해 자식 컴포넌트에 상태를 전달해야 하는 경우도 있습니다. 이러한 경우, React에서는 공통의 조상 컴포넌트에서 해당 state를 사용하는 자손 컴포넌트까지 props로 해당 state를 전달해 주어야 합니다. 이는 규모가 큰 프로젝트에서는 몇 번이고 props를 전달해야 하는 불편함을 낳을 수 있습니다. 그러나 redux를 사용하면, **root 레벨에서 모든 state를 관리**할 수 있고, 각 컴포넌트는 이를 **구독**하여 한 번에 가져다 쓸 수 있게 되어, **state의 컴포넌트 종속성으로부터 자유**로워질 수 있습니다.

2. FLUX 패턴



   Redux는 다음과 같이 FLUX 패턴을 따라 동작합니다.

   1. **Action**

      Action은 데이터를 변경하는 행위로, Dispatcher에 발생한 Action의 **Type**과 연관된 데이터인 **Payload**로 이루어진 객체를 전달합니다.

   2. **Dispatcher**

      데이터의 흐름을 관리하는 허브입니다. 전달 받은 Action Type에 따라 미리 Store에서 등록해 둔 Reducer 함수를 실행합니다. 복수의 state 간에 의존성이 존재할 때, 이를 관리하는 것 역시 Dispatcher의 역할입니다.

3. **Store**

   state의 저장소로, 각각의 state와 Reducer 함수가 존재합니다. Action Type에 따른 Reducer 함수를 Dispatcher에 등록하며, 이것이 실행되어 state가 변경되면 View에 알립니다.

4. **View(Components)**

   state가 변경되면 최상위 View에서, 해당 state를 구독하는 모든 자식 View에 전달합니다. state의 변화를 전달 받은 각각의 View는 화면을 리렌더링합니다. 또한, 패턴의 시작인 Action 역시 View에서 발생합니다.

3. 불변성

   Redux에서는 state의 값이 바뀌었는지 여부를 **참조값(주소)의 변경 여부**로 판단합니다. 따라서 불변성을 유지해야 의도대로 동작할 것입니다.

4. 단점

   redux는 오직 상태 관리를 위한 라이브러리입니다. 그 외의 기능이 필요한 경우, 다른 라이브러리를 추가해 사용해야 합니다. 예를 들면, reducer는 동기적으로 동작하기 때문에, 서버 데이터를 사용하기 위해서는 Redux-saga, Redux-thunk, Redux-toolkit 등과 같은 미들웨어를 사용해야 합니다. 이로 인해 코드가 복잡해지고, 보일러 플레이트가 비대해 질 수 있다는 단점이 있습니다.