3차시 스크립트

Recap

- ul에다가 자바스크립트 통해서 동적으로 생성해서(동적이란 말은 원래 런타임에 생성한다는 뜻. 웹에서는 용어가 다른 말들이 많은데, 빌드도 js에서는 binary로 바꾼다는 말이 아님 min파일을 만든다는 뜻) 동적이란건 여기서는 html에 없는 요소를 만들어낸다는 뜻 dom요소 조작의 일부.
- 여기서 function renderTodos나 addEventListner등은 바로 동작하는것이 아니라(선언 즉시 실행이 아니라) 실행했을 때 동작하는것. renderTodos를 실행해주는 코드가 마지막에 들어있고.. 실행되면 localStorage에서 갖다다가 todos라는 변수에 저장하고 dom에 있는 필요한 요소들을 document.getElementById로 갖다다가 그 안에 요소 추가하고, saveTodos로 localStorage에 있는 데이터 변경한다. todos변수는 페이지가 리로드 될때마다 사라지지만 localStorage에 있는 데이터는 불변이기 때문에 계속 다시 가져오는것
- 여기서는 동적으로 생성되는 요소들에 각각 eventListener를 달아줬는데, 사실 이건 좋은 방법은 아니고.. 상위 요소에만 이벤트 리스너 달고 event.target객체로 자식요소에 접근해서 이벤트 발생하는것 확인하도록 하는것이 성능이 더 좋다. 이것을 이벤트 위임(delegation)이라고 한다.
- form의 submit이 일어나면 페이지 리로딩이 발생하면서 renderTodos()가 다시 실행된다.
- 단순한 앱인데도 코드가 복잡해진다. 그래서 style을 .css로 분리하고, script도 js파일로 분리한다. 하지만 여기서 페이지가 많아진다면? 하나의 html에 여러개의 js파일을 갖다가 쓰기도 하고 css파일도 여러개 생기고.. 페이지가 많은것을 multi page app이라고 한다. 코드의 복잡성이 올라간다.
- 여기까지 했으면 기본은 다 했고 사실 굉장히 많은것을 만들 수 있다. 꼭 react를 배워야만 원하는것을 만들 수 있는건 아니다. 실제로 github은 굉장히 큰 웹앱이지만 react를 사용하지 않는다. web-component라는것을 사용하긴 하는데.. 다양한 라이브러리를 가져다가 사용하면서 많은 것들을 시도해보면 좋을것같다.
- ex)threejs, p3.js paper.js p5.js phaser.js

```
<!DOCTYPE html>
<html lang="ko">
<head>
  <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To doList</title>
    <style>
        body { margin: 20px; }
        #todo-form { margin-bottom: 20px; }
        #todo-list { list-style: none; padding: 0; }
        #todo-list li { padding: 10px; border-bottom: 1px solid #ccc; display: flex;
        .completed { text-decoration: line-through; color: gray; }
        button.delete { margin-left: auto; background-color: red; color: white; bor
    </style>
</head>
<body>
    <h1>투두리스트</h1>
    <form id="todo-form">
        <input type="text" id="todo-input" placeholder="할 일을 입력하세요" requi
        <button type="submit">추가</button>
    </form>
    <ul id="todo-list"></ul>

    <script>
        let todos = JSON.parse(localStorage.getItem("todos")) || [];

        const todoForm = document.getElementById("todo-form");
        const todoInput = document.getElementById("todo-input");
        const todoList = document.getElementById("todo-list");

        function saveTodos() {
            localStorage.setItem("todos", JSON.stringify(todos));
        }

        function renderTodos() {
            // 기존 리스트 초기화
            todoList.innerHTML = "";
            todos.forEach(todo => {
                const li = document.createElement("li");
                li.dataset.id = todo.id;

                // 완료 상태에 따른 클래스 추가
```

```
            if (todo.completed) {
                li.classList.add("completed");
            }

            // 투두 텍스트
            const span = document.createElement("span");
            span.textContent = todo.task;
            // 완료 상태 토글 이벤트
            span.addEventListener("click", () => {
                todo.completed = !todo.completed;
                saveTodos();
                renderTodos();
            });
            li.appendChild(span);

            // 삭제 버튼
            const delButton = document.createElement("button");
            delButton.textContent = "삭제";
            delButton.className = "delete";
            delButton.addEventListener("click", () => {
                // 클릭한 투두 삭제
                todos = todos.filter(item => item.id !== todo.id);
                saveTodos();
                renderTodos();
            });
            li.appendChild(delButton);

            todoList.appendChild(li);
        });
    }

    // 폼 제출 시 새로운 투두 추가
    todoForm.addEventListener("submit", (e) => {
        e.preventDefault();
        const task = todoInput.value.trim();
        if (task !== "") {
            // 고유 ID 생성 (타임스탬프 사용)
            const newTodo = {
```

```
                id: Date.now(),
                task: task,
                completed: false
            };
            todos.push(newTodo);
            saveTodos();
            renderTodos();
            todoInput.value = "";
        }
    });

    // 페이지 로드 시 투두 렌더링
    renderTodos();
  </script>
</body>
</html>
```

강의목표

✔️ ES6 주요 문법 이해하기
✔️ 비동기 코드 작성 방법 익히기
✔️ React의 필요성과 기본 개념 파악하기
✔️ React 프로젝트를 생성하고 폴더 구조 이해하기
✔️ React의 기본 컴포넌트를 만들고 렌더링하기

1. ES6 문법 및 javascript 심화

1.1 let과 const

- `let`: 변수 선언, 재할당 가능
- `const`: 상수 선언, 재할당 불가능

```
let name = "John";
name = "Doe"; // 가능

const age = 25;
age = 30; // 오류 발생
```

📌 `var`는 왜 사용하지 않을까?

- `var`는 블록 스코프를 가지지 않아 의도치 않은 변수 재정의가 발생할 수 있음.
- `let`과 `const`를 사용하면 보다 안전한 코드 작성 가능.

1.2 화살표 함수

- `function` 키워드 없이 함수를 간결하게 표현
- `this`를 기존 함수와 다르게 바인딩함

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5
```

📌 화살표 함수 vs 일반 함수

- 화살표 함수는 `this`를 상위 스코프에서 가져오기 때문에, 클래스 메서드에서 `this`를 직접 바인딩할 필요 없음.

1.3 구조 분해 할당

- 배열과 객체에서 값을 쉽게 추출하는 문법

```
const person = { name: "Bob", age: 30 };
const { name, age } = person;
console.log(name, age); // Bob 30

const numbers = [1, 2, 3];
const [first, second] = numbers;
console.log(first, second); // 1 2
```

📌 기본값 설정 가능

```
const { city = "Seoul" } = {};
console.log(city); // Seoul
```

1.4 Spread Operator

JavaScript의 스프레드 연산자(spread operator)는 객체나 배열의 요소를 펼치는 데 사용되는 기능입니다. 주로 배열이나 객체를 복사하거나 합치는 데 유용합니다. 스프레드 연산자는 세 개의 점(...)으로 표현됩니다.

배열에서의 사용

1. **배열 복사**:

```
const arr = [1, 2, 3];
const newArr = [...arr]; // [1, 2, 3]
```

2. **배열 합치기**:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]
```

객체에서의 사용

1. **객체 복사**:

```
const obj = { a: 1, b: 2 };
const newObj = { ...obj }; // { a: 1, b: 2 }
```

2. **객체 합치기**:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const merged = { ...obj1, ...obj2 }; // { a: 1, b: 3, c: 4 }
```

주의사항

스프레드 연산자는 얕은 복사를 수행합니다. 즉, 객체나 배열 내에 다른 객체나 배열이 포함되어 있을 경우, 원본 객체의 참조를 유지하게 됩니다.

1.5 this와 this 바인딩

this란?

`this`는 현재 실행 컨텍스트(context)를 나타내는 특별한 키워드야.

즉, 함수나 객체 내에서 `this`를 참조하면, 그 때의 실행 문맥에 따라 다른 객체를 가리키게 돼.

this는 python에서의 self와 같다.

this 바인딩. 바인딩이란 this가 어떤것을 가리키도록 하는것. bind는 묶다라는 뜻

this 바인딩

- **일반 함수 호출:**
  - 일반 함수 내에서의 `this`는 보통 전역 객체(window)를 가리키게 돼.

```
function show() {
    console.log(this);
}
show(); // 브라우저에서는 window 객체, strict 모드에서는 undefined
```

- **객체의 메서드 호출:**
  - 객체의 메서드 내에서의 `this`는 해당 메서드를 호출한 객체를 가리켜.

```
const obj = {
    value: 42,
    getValue: function() {
        console.log(this.value);
    }
};
obj.getValue(); // 42, 여기서 this는 obj를 참조
```

- **생성자 함수에서의 this:**
  - 생성자 함수에서의 `this`는 새로 생성되는 인스턴스를 가리켜.

```
function Person(name) {
    this.name = name;
}
const person1 = new Person("Bob");
console.log(person1.name); // Bob
```

- **화살표 함수에서의 this:**

- 화살표 함수는 자신만의 `this`를 가지지 않고, 외부(상위 스코프)의 `this`를 그대로 사용해.
- 따라서, 콜백 함수나 이벤트 핸들러 내에서 상위 컨텍스트의 this가 필요한 경우 유용하다.

```
const obj2 = {
    value: 100,
    getValue: function() {
        setTimeout(() => {
            console.log(this.value); // 여기서 this는 obj2를 참조
        }, 100);
    }
};
obj2.getValue(); // 100
```

this 바인딩의 종류와 조작

다음과 같이 this를 명시적으로 조작할 수도 있어.

- **call():**
  - 함수를 호출하면서, this의 값을 지정할 수 있음.

```
function greet() {
    console.log(`안녕, ${this.name}`);
}
const user = { name: "Alice" };
greet.call(user); // 안녕, Alice
```

- **apply():**
  - call과 비슷하지만, 두 번째 인자로 배열을 전달하여 매개변수를 넘긴다.

```
function introduce(age, city) {
    console.log(`안녕, ${this.name}. ${age}살이고 ${city}에 살아.`);
```

```
}
introduce.apply(user, [25, "서울"]);
```

- **bind():**
  - 새로운 함수를 반환하며, 이 함수는 미리 지정된 this를 갖는다.

```
const boundGreet = greet.bind(user);
boundGreet(); // 안녕, Alice
```

> 예상 질문:
>
> **Q:** 왜 화살표 함수는 자신의 this를 가지지 않을까?
>
> **A:** 화살표 함수는 상위 스코프의 this를 그대로 사용하여, 콜백 함수나 메서드 내부에서 의도치 않은 this 변경 문제를 방지하기 위해 설계되었어. 객체의 메서드를 정의할 때는 일반 함수를 사용하는 것이 적절할 수 있다.

2. 비동기 JavaScript

2.1 Promise

- 비동기 작업을 처리하는 객체

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Data received"), 2000);
});

fetchData.then(data => console.log(data)); // 2초 후 "Data received" 출력
```

2.2 async/await

- 비동기 코드를 동기 코드처럼 작성 가능

```
async function getData() {
  const data = await fetchData;
  console.log(data);
}

getData();
```

📌 `try-catch`로 에러 처리

```
async function fetchDataWithError() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data", error);
  }
}
```

3. React의 필요성

2000년대 후반부터 웹 앱은 점점 복잡해졌다. UI가 동적으로 변하는 기능이 점점 복잡해졌고 DOM조작을 편하게 하기 위해서 jQuery와 같은 직접 DOM을 조작하는 방식을 사용했다. jQuery는 간편한 API로 DOM을 조작할 수 있게 해주면서 굉장히 많이 사용되었다. 아직도 jQuery를 사용하는 사이트가 많다. 동작하면 건들지말라는게 개발의 철칙이기 때문이다. jQuery덕분에 CSS를 조작하거나, 애니메이션 등을 넣는것이 쉬워졌다.

하지만 jQuery는 성능이 나쁘다. DOM조작을 위한 라이브러리이지만 그냥 javascript (vanilaJS)를 사용하는것과 비교해서 수백배 느리기도 하다. 이걸 오버헤드가 크다고 한다. 작은 동작을 하는데 기능이 많은 도구를 쓰면 발생하는 문제이다.

렌더링 비효율성

jQuery의 속도문제 뿐만 아니라 다양한 문제점이 있었는데, 매번 DOM을 다시 그리는 것은 비효율적이라는 것이다. 1000개의 요소가 있다고 했을때, 한개만 바뀌었는데 나머지까지 전부 다시 그리는 것은 비효율적이다. React는 Virtual Dom을 사용해서 변경된 부분만 업데이트할 수 있도록 해준다.

virtual dom은 변경사항을 메모리 상에서 먼저 처리하고, 실제 DOM업데이트는 최소화한다.

유저경험상으로도 안좋은데, 화면에 작은 변화만 적용하려고 해도 페이지를 새로고침해야하기 때문에 깜빡거리는 느낌이 든다

코드의 복잡성 증가

각 요소를 동적으로 만들기 위해서 createElement하고, 이벤트리스너 붙이고.. 이런식으로는 요소들을 관리하기가 어렵다. React는 컴포넌트 기반으로 UI를 쪼개서 관리하므로 유지보수가 쉬워진다.

코드를 컴포넌트 단위로 나눠놓으면 재사용하기도 쉽고 기능별로, 페이지별로 필요한 컴포넌트를 분리할 수있다

상태관리의 어려움

상태란 UI와 연결되어있는 변수이다. 상태가 변경되면 UI도 변경되어야 한다

todos배열이 변경되면 renderTodos()를 호출해서 화면을 다시 그려줘야한다. 하지만 함수 호출을 빼먹는다면 UI는 todos의 값과 불일치하게 된다. 즉 상태의 일관성을 유지하기가 어렵다.

todos배열이 변경되는 경우가 다양하다면? renderTodos를 호출해야하는 경우가 다양하다면? 상태관리가 무척어렵다. React는 상태와 UI를 효율적으로 연결시켜준다.

즉, react의 가장 큰 기능은 state가 변경되면 자동으로 컴포넌트를 rerendering해주는것! 그래서 DOM을 직접 변경하지 않아도 되게 해주는 것!

나머지는 컴포넌트로 나누고.. virtual dom으로 성능향상하고.. 그런건 부수적인거다

virtual dom은 정말 빠른가?

사실 virtual dom은 항상 vanilajs보다 빠른것은 아니다. react를 사용하면 오버헤드가 발생하고.. 대규모 어플리케이션에서는 효율적인 경우가 있다. virtual dom은 state의 업데이트가 발생했을때 즉각적으로 적용하는것이 아니라, 여러번 업데이트 되더라도 실제 dom에 변경사항 적용하는것을 중간 layer에서 메모리상에서만 적용해서 실제 dom변경은 최소화한다. setState함수가 호출되는 경우에는 실제 변경사항이 적용되는것은 render()가 호출된 이후이다.

virtual dom의

✅ React의 장점

- **컴포넌트 기반 개발**: 재사용성과 유지보수성이 향상됨
- **가상 DOM**: 실제 DOM을 최소한으로 갱신하여 성능 최적화

- **상태 관리**: UI 상태(state)를 손쉽게 관리 가능

3.2 기본 개념

- **컴포넌트(Component)**: UI의 재사용 가능한 단위
- **JSX(JavaScript XML)**: HTML과 유사한 문법을 사용해 UI를 정의

📌 JSX 예제

```
const element = <h1>Hello, World!</h1>;
ReactDOM.render(element, document.getElementById("root"));
```

슬라이드 2: jQuery에서 React로의 전환 배경

웹 애플리케이션이 점점 복잡해지면서 단순한 DOM 조작을 넘어 복잡한 상태 관리가 필요하게 되었습니다. jQuery는 당시 인기 있는 도구였지만 몇 가지 한계에 부딪혔습니다:

1. **명령형 프로그래밍 방식**: jQuery는 "어떻게(how)" 화면을 변경할지 일일이 지시해야 했고, 이는 복잡한 UI 상태 관리를 어렵게 만들었습니다.
2. **코드 재사용성 부족**: 동일한 기능을 여러 곳에서 사용하려면 코드를 중복해야 했습니다.
3. **유지보수 어려움**: 애플리케이션 규모가 커질수록 코드 관리가 어려워졌습니다.
4. **성능 병목 현상**: 대규모 애플리케이션에서 잦은 DOM 조작은 성능 저하를 가져왔습니다.

페이스북은 실시간 업데이트와 복잡한 UI 상태를 효율적으로 관리할 수 있는 새로운 접근 방식이 필요했고, 이것이 React 개발의 동기가 되었습니다.

슬라이드 3: React의 탄생

React는 2011년 페이스북 내부에서 개발이 시작되었습니다. 그는 "데이터가 변경될 때마다 전체 앱을 다시 렌더링하면 어떨까?"라는 질문에서 출발했습니다. = 데이터와 UI를 일치시키기 위해서. 이는 기존의 DOM 조작 방식과는 완전히 다른 접근법이었습니다.(기존에는 명령형, 지금은 선언형)

2013년 JSConf에서 React가 처음 공개되었을 때, 많은 개발자들이 의아해했습니다. HTML을 JavaScript 안에 작성하는 JSX 문법은 당시 웹 개발 관행과 달랐기 때문입니다. 그러나 페이스북 뉴스피드와 인스타그램 같은 복잡한 UI를 가진 애플리케이션의 문제를 해결하는 과정에서 React의 가치가 입증되었습니다.

슬라이드 4: React의 개발 철학

React는 몇 가지 핵심 철학을 바탕으로 설계되었습니다:

1. **선언적 프로그래밍**: "어떻게(how)" 화면을 그릴지가 아니라 "무엇을(what)" 보여줄지에 집중합니다. 개발자는 UI가 어떻게 보여야 하는지 선언하고, React가 실제 DOM 업데이트를 처리합니다.
2. **컴포넌트 기반 아키텍처**: UI를 독립적이고 재사용 가능한 조각(컴포넌트)로 분리하여 복잡한 인터페이스를 관리하기 쉽게 만듭니다.
3. **단방향 데이터 흐름**: 상위 컴포넌트에서 하위 컴포넌트로 데이터가 흐르는 단방향 구조로, 예측 가능한 상태 관리를 가능하게 합니다.
4. **"Learn Once, Write Anywhere"**: React의 패러다임을 한 번 배우면 웹(React), 모바일(React Native), 데스크톱(Electron + React) 등 다양한 플랫폼에서 활용할 수 있습니다.

슬라이드 5: React의 주요 기능

React의 핵심 기능들은 다음과 같습니다:

1. **JSX**: JavaScript 내에서 HTML 형태의 마크업을 작성할 수 있는 문법 확장입니다. 코드의 구조와 시각적 표현을 함께 볼 수 있어 직관적입니다.

```
const element = <h1>Hello, world!</h1>;
```

2. **컴포넌트**: UI의 독립적인 조각으로, 자체 로직과 렌더링 방식을 가집니다. 함수형 컴포넌트와 클래스 컴포넌트 두 가지 형태로 작성할 수 있습니다.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

3. **Props**: 부모 컴포넌트에서 자식 컴포넌트로 데이터를 전달하는 방식입니다.

```
<Welcome name="Sara" />
```

4. **State**: 컴포넌트 내에서 관리되는 데이터로, 변경될 수 있으며 변경 시 컴포넌트가 다시 렌더링됩니다.

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

5. **생명주기 메서드/Hooks**: 컴포넌트의 생성, 업데이트, 제거 과정에서 특정 코드를 실행할 수 있게 합니다.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);
```

6. **이벤트 처리**: DOM 이벤트를 캡슐화하여 일관된 방식으로 처리합니다.

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

슬라이드 6: Virtual DOM 개념

Virtual DOM은 React의 핵심 개념 중 하나로, 실제 DOM의 가벼운 JavaScript 복사본입니다. 웹 브라우저의 DOM 조작은 상대적으로 느린 작업인데, React는 이 문제를 해결하기 위해 Virtual DOM을 사용합니다.

작동 방식은 다음과 같습니다:

1. 데이터가 변경되면 React는 전체 UI를 Virtual DOM에 렌더링합니다.
2. 이전 Virtual DOM 스냅샷과 새로운 Virtual DOM을 비교합니다.
3. 실제로 변경된 부분만 식별하여 실제 DOM에 적용합니다.

이 과정을 통해 불필요한 DOM 조작을 최소화하고 성능을 최적화합니다.

근데 결국 가상돔도 중간과정이 들어가는거고 dom조작 자체가 빨라지는건 아님

슬라이드 7: Diffing 알고리즘

Diffing 알고리즘은 두 Virtual DOM 트리를 비교하여 변경된 부분을 효율적으로 식별하는 방법입니다. 이 과정은 이론적으로 $O(n^3)$ 시간 복잡도를 가질 수 있지만, React는 두 가지 핵심 가정을 통해 $O(n)$ 시간 복잡도로 최적화했습니다:

1. **서로 다른 타입의 두 요소는 다른 트리를 생성한다**: 예를 들어 `<div>`가 `<span>`으로 변경되면, React는 이전 트리를 버리고 새로운 트리를 구축합니다.
2. **개발자는 key prop을 통해 안정적인 항목을 식별할 수 있다**: 리스트 렌더링 시 각 항목에 고유한 `key`를 지정하면 React가 항목의 이동, 추가, 삭제를 효율적으로 처리할 수 있습니다.

```
{items.map(item => (
  <ListItem key={item.id} item={item} />
))}
```

이 알고리즘은 리스트의 마지막에 항목을 추가하는 경우 매우 효율적이지만, 중간에 항목을 삽입하면 성능이 저하될 수 있습니다. 따라서 고유하고 안정적인 key 설정이 중요합니다.

슬라이드 8: Reconciliation (재조정)

Reconciliation은 Virtual DOM의 변경 사항을 실제 DOM에 효율적으로 적용하는 과정입니다. 이 과정은 두 단계로 나뉩니다:

1. **렌더링 단계(Render Phase)**: 컴포넌트를 호출하고 변경 사항을 계산합니다. 이 단계는 비동기적으로 수행될 수 있으며, React는 필요에 따라 이 작업을 중단하거나 우선순위를 조정할 수 있습니다.
2. **커밋 단계(Commit Phase)**: 계산된 변경 사항을 실제 DOM에 적용합니다. 이 단계는 항상 동기적으로 수행되어 UI가 일관된 상태를 유지합니다.

React 16부터 도입된 Fiber 아키텍처는 렌더링 작업을 작은 단위로 나누어 우선순위를 지정하고, 필요에 따라 작업을 중단하고 재개할 수 있게 해주어 더 나은 사용자 경험을 제공합니다.

state가 변경될때마다 diffing과 reconciliation이 수행되는것은 아니고 batch update한다 설명했던것처럼! 여러개의 상태변화를 묶어 한번에 처리함

슬라이드 9: React 프로젝트 폴더 구조

React 프로젝트는 일반적으로 다음과 같은 구조를 가집니다:

기본 Create React App 구조:

```
my-app/
├── node_modules/      # 설치된 패키지
├── public/            # 정적 파일
│   ├── index.html     # 기본 HTML 템플릿
│   └── favicon.ico    # 파비콘
├── src/               # 소스 코드
│   ├── App.js         # 루트 컴포넌트
│   ├── index.js       # 진입점
│   ├── components/    # 재사용 컴포넌트
│   └── assets/        # 이미지 등 자원
```

```
├── package.json      # 프로젝트 설정
└── README.md         # 문서
```

슬라이드 10: jQuery vs React 비교

jQuery와 React는 근본적으로 다른 패러다임을 가지고 있습니다:

패러다임:

- jQuery: 명령형(Imperative) - DOM을 직접 선택하고 조작합니다.
- React: 선언형(Declarative) - 원하는 UI 상태를 선언하면 React가 DOM 조작을 처리합니다.

코드 구조:

- jQuery: 절차적 코드 흐름으로, 단계별 DOM 조작 명령을 직접 작성합니다.
- React: 컴포넌트 기반으로, UI를 재사용 가능한 조각으로 나누어 구성합니다.

상태 관리:

- jQuery: DOM 요소의 상태를 직접 변경하고 수동으로 동기화해야 합니다.
- React: 상태(state)가 변경되면 자동으로 UI가 업데이트됩니다.

예시 코드 비교:
jQuery:

```
$("#counter").text("0");
$("#increment").on("click", function() {
  const count = parseInt($("#counter").text());
  $("#counter").text(count + 1);
});
```

React:

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <span>{count}</span>
      <button onClick={() => setCount(count + 1)}>
        Increment
```

```
      </button>
    </div>
  );
}
```

슬라이드 11: React의 장점

React가 많은 개발자와 기업에 채택된 이유는 다음과 같은 장점이 있기 때문입니다:

1. **선언적 UI**: 코드가 더 예측 가능하고 디버깅하기 쉽습니다. 원하는 결과를 선언하면 React가 그것을 실현하는 방법을 처리합니다.
2. **컴포넌트 재사용**: UI를 독립적인 조각으로 나누어 개발 효율성을 높이고 일관성을 유지할 수 있습니다.
3. **Virtual DOM**: 최적화된 렌더링으로 성능을 개선합니다. 특히 복잡한 UI나 자주 업데이트되는 애플리케이션에서 효과적입니다.
4. **단방향 데이터 흐름**: 상태 변화를 추적하기 쉽고 예측 가능한 코드를 작성할 수 있습니다.
5. **강력한 생태계**: Redux, React Router, Styled Components 등 다양한 라이브러리와 도구가 있어 개발을 더 효율적으로 만듭니다.

4. 프로젝트 생성

4.1 Vite 사용

```
npm create vite@latest my-app --template react
cd my-app
npm install
npm run dev
```

4.2 컴포넌트 생성 (App.jsx)

```
import React from "react";

function App() {
  return <h1>Hello, React!</h1>;
}
```

```
export default App;
```

4.3 렌더링 (index.js)

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(<App />);
```

---

5. 실습: Todolist 만들기

Svelte

React는 라이브러리고 svelte는 컴파일러다

react사이트는 react의 기능을 담은 js파일을 사용해서 동작한다. 빌드할때 react라이브러리의 코드를 포함해서 js로 내놓는다는것. 사이트가 실행될 때 코드에 있는 템플릿과 데이터를 읽어서 화면에 그린다. react 사용해서 작성한 js코드를 브라우저가 가상돔으로 해석해서 메모리에서 구현하고, 최종적으로 dom에 적용하는것. 즉 런타임에 가상돔만들고..하는것

svelte는 컴파일 타임에 실행함. 사이트가 배포되기 전에 미리 그 과정을 해놓음. 어떤 데이터가 바뀌었을때 어떻게 변경될지를 미리 계산해놓고 포함한다. 컴파일과정에서 이미 끝남

Fiber 아키텍처란?

Fiber 아키텍처는 React 16부터 도입된 새로운 내부 구조로, 주로 **비동기 렌더링**과 **업데이트 스케줄링**을 더 유연하게 하기 위해 개발되었습니다.

렌더링 작업을 작은 조각으로 나누고, 우선순위를 부여해 중요한 작업을 먼저 처리하는 방식으로, UI가 끊기지 않고 부드럽게 동작하도록 도와줍니다.

- **작업을 작은 단위로 쪼개기:**

  기존에는 한 번에 큰 작업을 처리했지만, Fiber는 작업을 여러 작은 단위(작업 단위, fiber)로 쪼개어 실행합니다.

- **우선순위에 따른 스케줄링:**

각 작업 단위에 우선순위를 부여해, 중요한 작업(예: 사용자 입력 처리)과 덜 중요한 작업(예: 백그라운드 데이터 업데이트)을 구분합니다.

높은 우선순위 작업이 있을 때는 현재 작업을 잠시 중단하고 중요한 작업을 먼저 처리할 수 있습니다.

- **유연한 작업 처리:**

  이로 인해 브라우저가 장시간 동안 긴 작업에 묶여 있지 않게 되어, 사용자 경험이 더욱 부드럽고 응답성이 높아집니다.