React 프론트엔드 개발자를 위한 CS 지식 심화 정리

---

1. HTML/CSS 기본 및 렌더링 메커니즘

HTML은 웹 페이지의 구조를 정의하는 마크업 언어다. `<div>`, `<header>`, `<section>` 같은 태그를 사용해 콘텐츠를 의미 단위로 나눌 수 있다. 특히 시맨틱 태그는 검색 엔진이 페이지 구조를 이해하는 데 도움을 주며, 스크린 리더 같은 접근성 도구에도 유리하다.

CSS는 시각적인 스타일을 지정하기 위한 언어로, 선택자를 통해 특정 요소를 선택하고 색상, 여백, 레이아웃 등을 설정한다. 스타일 우선순위와 박스 모델을 정확히 이해하지 못하면, 원하지 않는 스타일 충돌이나 레이아웃 깨짐이 발생할 수 있다.

렌더링 메커니즘은 HTML을 파싱해 DOM 트리를 만들고, CSS를 파싱해 CSSOM을 만든 뒤 이 둘을 결합해 Render Tree를 생성한다. 이후 Layout → Paint → Composite 단계를 통해 브라우저 화면에 그려진다. 이 흐름을 이해하면 reflow/repaint 발생 시점과 성능 최적화 포인트를 알 수 있다.

---

2. Closure / Scope / Hoisting

스코프는 변수나 함수가 유효한 범위를 의미한다. `var`는 함수 스코프, `let`/`const`는 블록 스코프를 가지며, 전역과 지역 스코프를 명확히 구분해야 변수 충돌이나 예기치 않은 동작을 방지할 수 있다.

클로저는 함수가 선언될 당시의 외부 렉시컬 환경을 기억해 참조할 수 있는 구조다. 상태 유지를 위한 함수나 비동기 로직에서 주로 활용되며, React에서 useCallback, useEffect 안에서 stale closure 문제가 생길 수 있다.

호이스팅은 변수와 함수 선언이 코드 실행 전에 스코프의 최상단으로 끌어올려지는 JS의 동작 방식이다. `var`는 undefined로 초기화되며 호이스팅되고, `let`/`const`는 TDZ로 인해 선언 전 참조 시 오류가 발생한다. 이 특성을 몰라 발생하는 버그가 많기 때문에 주의가 필요하다.

---

3. Event Loop & Call Stack

JS는 싱글 스레드 언어이지만 비동기 처리를 위해 Event Loop 구조를 사용한다. Call Stack은 함수 실행 흐름을 추적하는 구조이며, 함수가 호출되면 스택에 쌓이고, 실행이 끝나면 제거된다.

비동기 작업은 Task Queue에 등록된다. setTimeout 같은 Macro Task는 일반 큐에, Promise.then 같은 Micro Task는 Microtask Queue에 쌓이며, Micro Task가 우선 처리된다.

Event Loop는 Call Stack이 비었는지를 확인하고, 비었다면 큐에서 Task를 꺼내 실행한다. 이 구조를 모르면 비동기 로직이 왜 특정 순서로 실행되는지를 설명할 수 없다.

---

4. this 바인딩

this는 함수가 호출되는 방식에 따라 달라진다. - 일반 함수: 전역 객체(window), strict mode에서는 undefined - 객체 메서드: 해당 객체 - 생성자 함수: 생성된 인스턴스 - 화살표 함수: 선언 당시의 this를 계승

React에서는 클래스형 컴포넌트에서 this 바인딩 문제가 자주 발생하며, `this.method = this.method.bind(this)` 같은 코드가 필요하다. 함수형 컴포넌트에서는 주로 화살표 함수를 사용하여 this 이슈를 피한다.

---

5. Promise / async-await / Microtask Queue

Promise는 비동기 작업의 상태와 결과를 표현하는 객체다. `.then`과 `.catch`를 통해 결과와 에러를 처리할 수 있다.

async/await은 Promise를 더 간결하게 작성할 수 있게 해주는 문법으로, 동기 코드처럼 작성할 수 있어 가독성이 좋다. try/catch와 함께 사용하면 에러 핸들링도 명확해진다.

Promise의 후속 작업은 Microtask Queue에 들어가며, call stack이 비면 즉시 실행된다. setTimeout보다 먼저 실행된다는 특징 때문에 순서 제어가 필요한 상황에서 중요한 차이로 작용한다.

---

6. DOM 구조 및 조작 원리

DOM은 HTML 문서를 트리 형태로 구조화한 객체 모델이다. 자바스크립트를 통해 DOM을 직접 조작할 수 있고, `.querySelector`, `.appendChild` 등을 통해 노드를 탐색/추가/삭제할 수 있다.

React에서는 직접 DOM을 조작하는 대신 Virtual DOM을 활용해 변경 사항을 계산하고 최소한의 조작만 수행한다. 이를 통해 효율적인 UI 업데이트가 가능하다.

---

7. HTTP/HTTPS 및 RESTful API

HTTP는 클라이언트-서버 간 요청/응답 구조의 텍스트 기반 통신 프로토콜이다. HTTPS는 여기에 SSL/TLS 암호화를 더한 형태로 보안을 강화한 프로토콜이다.

RESTful API는 자원을 URI로 표현하고, GET/POST/PUT/DELETE 같은 HTTP 메서드를 통해 자원에 대한 작업을 수행한다. 일관된 규칙에 따라 설계되기 때문에 API 사용성과 유지보수성이 높다.

---

8. 브라우저 렌더링 과정

HTML과 CSS가 파싱되어 DOM과 CSSOM이 만들어지고, 이를 기반으로 Render Tree가 생성된다. 이후 Layout(크기/위치 계산) → Paint(픽셀 변환) → Composite(레이어 합성) 과정을 거쳐 사용자 화면에 표시된다.

이 과정 중 JavaScript는 렌더링을 막을 수 있기 때문에 script 태그의 위치나 async/defer 사용 여부가 성능에 영향을 준다.

---

9. 웹팩(Webpack) / 바벨(Babel)

Webpack은 JS, CSS, 이미지 등 모든 자원을 하나의 번들로 묶어주는 모듈 번들러다. Tree-shaking, Code Splitting 같은 기능을 통해 최적화된 번들을 만들 수 있다.

Babel은 최신 자바스크립트 문법(예: ES6, JSX)을 하위 브라우저에서도 동작하도록 변환해주는 트랜스파일러다. React에서 JSX를 사용하려면 Babel 설정이 필수다.

---

10. 상태 관리 원리 (Redux, Context API)

Redux는 전역 상태 관리를 위한 라이브러리로, 상태는 store에 저장되고 dispatch(action)를 통해 reducer로 상태를 변경한다. 상태는 불변성을 유지해야 하며, 이를 통해 변경 감지를 효율적으로 수행할 수 있다.

Context API는 React 내장 기능으로, props drilling 없이 하위 컴포넌트에 상태를 전달할 수 있다. useContext 훅과 Provider/Consumer 패턴으로 구성되며, 전역 상태 관리가 간단한 프로젝트에 적합하다.

---

11. CSR vs SSR vs SSG 개념

- CSR(Client Side Rendering): JS가 브라우저에서 실행되어 렌더링됨. 초기 로딩은 느리지만 상호작용이 빠름.
- SSR(Server Side Rendering): 서버에서 HTML을 생성해 전달. 초기 렌더링이 빠르고 SEO에 강점이 있음.
- SSG(Static Site Generation): 빌드 시 HTML을 만들어 배포. 정적인 콘텐츠에 적합하며 속도가 빠름.

Next.js는 이 3가지 렌더링 방식을 모두 지원하며, 페이지 성격에 따라 전략적으로 혼합 사용할 수 있다.

---

12. 이벤트 버블링 캡쳐링

이벤트는 DOM 구조에서 아래에서 위(버블링) 또는 위에서 아래(캡쳐링) 방향으로 전파된다. 기본적으로는 버블링이 적용되며, 캡쳐링을 사용하려면 addEventListener의 세 번째 인자를 true로 설정해야 한다.

React에서는 SyntheticEvent 시스템을 사용해 이벤트를 버블링 방식으로 처리하며, 이벤트 위임 구현도 가능하다. 이벤트 전파를 막고 싶을 경우 `e.stopPropagation()`을 사용한다.

---

13. CORS 정책 개념

CORS(Cross-Origin Resource Sharing)는 브라우저가 보안을 위해 다른 origin에서 요청을 막는 정책이다. 서버가 `Access-Control-Allow-Origin` 헤더를 설정하지 않으면 브라우저가 응답을 차단한다.

프론트엔드 개발 시 API 호출이 막히는 경우가 많아, CORS 에러의 원인과 해결 방법(proxy 설정 등)을 정확히 이해하고 있어야 한다.

---

14. 자바스크립트 실행문맥

실행 컨텍스트는 자바스크립트 코드가 실행될 때 변수, 함수, this, 스코프 정보 등을 저장하는 객체다. 전역 실행 컨텍스트, 함수 실행 컨텍스트가 있으며, 이들이 쌓여 Execution Stack을 이룬다.

컨텍스트 안에는 Variable Environment, Lexical Environment, this 바인딩 정보 등이 포함되며, 변수 참조 및 실행 흐름을 이해하는 데 핵심 개념이다.

---

15. 자바스크립트가 병렬로 처리하는 방법

자바스크립트는 싱글 스레드 기반이지만 Web API, Event Loop 구조를 활용해 비동기 작업을 병렬적으로 처리한다. `setTimeout`, `fetch`, `Promise`, `Web Worker` 등이 병렬 처리의 대표 예다.

Web Worker는 별도의 스레드에서 동작하며, 무거운 계산 로직을 분산시켜 메인 스레드의 부하를 줄일 수 있다. React에서 대량 연산을 처리할 경우 Worker 사용을 고려해볼 수 있다.

---

16. 리액트 상태 불변성

React는 상태의 변경 여부를 얕은 비교(shallow equality)로 판단하므로, 상태를 직접 수정하면 참조가 바뀌지 않아 변경을 감지하지 못한다.

예를 들어, 배열에 값을 push하면 원본 배열이 바뀌지만 참조는 그대로이므로 리렌더링이 일어나지 않는다. 따라서 항상 `...spread` 또는 `concat`, `filter` 등을 사용해 새로운 객체나 배열을 만들어 상태를 변경해야 한다.

```
// ❌잘못된 예시
state.todos.push(newTodo);

// ✅올바른 방식
setTodos([...state.todos, newTodo]);
```

이 원칙을 지키면 React의 성능 최적화와 컴포넌트 재사용성이 올라가며, 버그 발생 가능성도 줄어든다.